University of Bristol


DEPARTMENT OF COMPUTER SCIENCE

# Concurrent and Distributed Functional Systems

Eleni Spiliopoulou

---

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Doctor of Philosophy in the Faculty of Engineering

September 1999 | CSPHD

**Abstract**

This thesis presents the *Brisk Machine* [54], a machine for executing functional languages, designed to be simple and flexible to support a number of run-time execution models for the Brisk compiler. Design considerations have been made to support dynamic loading, deterministic concurrency [23, 51], distribution, debugging tools and logic programming [76]. To achieve this, the compiler's intermediate language, the Brisk Kernel Language `BKL` is simpler than the `STG` language [100], as evaluation, extension and optimisation issues are relegated to special built-in functions. Moreover, function calls are saturated, as any function has an known arity and in every call to it, is applied to the right number of arguments, which makes the machine dynamic and supports *dynamic loading*.

To incorporate deterministic concurrency in Brisk, we present the Brisk monadic framework [55]. Its main innovation is to allow actions on state components. This is a key issue which enables *state splitting*, a technique which assigns to each new thread a part of the state, a *substate*, to act upon. Distinct concurrent threads are restricted to access disjoint substates. Thus, non-deterministic effects that arise when threads mutate the same state are avoided.

To extend the expressiveness of deterministic concurrency, a *deterministic form of communications* has been introduced, where input from several threads is merged deterministically. The merging is based on hierarchical timestamping on messages from different sources and allows messages to be merged in a consistent temporal order.

A model of distribution [53] is presented, as incorporated in the Brisk runtime system, as a natural consequence of deterministic concurrency. The abstract model of this distributed system uses a uniform access memory model to provide a global memory view. This, combined with explicit thread based concurrency and explicit representation of demand enables the trustable distribution of reactive systems across different systems. Since referential transparency is preserved, any pattern of distribution has no effect on the semantics of the

functional program. Computation, including both code and data, can be moved in response to demand.

Στούς γονεῖς μου Στέφανο καί Μάρθα

# Acknowledgements

First, I would like to thank my supervisor Dr. Ian Holyer for his guidance, enthusiasm and support throughout this thesis.

I also thank Neil Davies for his participation in the Functional Programming Group, the members of the Bristol Functional Programming Group Chris Dornan, Alastair Penney, Hussein Pehlivan and the members of the Declarative Systems Group.

I am also grateful to the Soroptimist International Organisation for funding this thesis.

Finally, I would like to thank my family and friends for all the support and encouragement during the course of this thesis, which has been proven invaluable.

# Declaration

The work in this thesis is the independent and original work of the author, except where explicit reference to the contrary has been made. No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other University or Institution of Education.

Bristol, 30/08/1999

Eleni Spiliopoulou

# Contents

# List of Figures

# Chapter 1

# Introduction

It is well understood that functional languages have interesting properties which, provided that they are well developed and supported with tools and professional education, would have a beneficial effect on the creation of large software systems. Based on the belief that the properties of functional languages allow programs to be well distributed, the challenge has been to provide a theory and a tool which could be used in order to extend the functional programming paradigm into different domains such as concurrency and distribution. This thesis forms part of the Brisk Project[1] which has been set up aiming to provide concurrent and distributed working environments where the properties of functional languages, notably referential transparency, are not compromised.

## 1.1 The Theory

### 1.1.1 Deterministic Concurrency

Concurrency introduces non-determinism as a consequence of timing dependencies during process creation and communication. Processes are created so that they share the same address space which authorises non-deterministic effects. Furthermore, processes communicate via non-deterministic primitives. As a result, concurrent systems inherently exhibit non-deterministic behaviour, which can manifest itself as undesired behaviour under certain accidental cir-

[1] `Brisk` stands for Bristol Haskell [5] compiler

cumstances of timing. This adds an extra burden of complexity onto the programmer, making it very difficult to assert that concurrent systems are reliable and resilient. (*Chapter 2*)

Additionally, any attempt to add non-deterministic features directly into a functional language destroys referential transparency and thus, equational reasoning cannot be applied [27]. This implies that cleaner models of concurrency are still required, at least for purely functional languages, in order to cope with these problems. (*Chapter 5*)

One such model of concurrency is *deterministic concurrency* [51, 23]. This extends the demand driven model, used as the operational basis of functional languages, into one with multiple independent demands in the form of threads that do not share any state. To avoid the non-deterministic effects that arise during thread creation, threads are created so that they are independent. This is in contrast to the traditional view of concurrency where threads share the same address space, and thus they are not independent. To avoid the non-deterministic effects that arise during thread communication because of the need for a non-deterministic merge, threads communicate only via values they produce as results. This form of concurrency has been initially introduced in Haskell 1.2 on top of the stream `I/O` model. Its principles together with good design practices have been proven expressive enough to provide a complete single-user multi-tasking working environment including shells, file managers and other single-user operating system features. (*Chapter 5*)

In this thesis, deterministic concurrency is built into Haskell on top of the monadic `I/O` mechanism [106][2]. This has the form of independent threads that act on disjoint parts of the `I/O` state, also called *substates*. Each thread splits at creation part of the state to act upon. Distinct concurrent threads are restricted to access disjoint substates so that non-determinism which arises during thread creation does not occur. (*Chapters 5, 6*)

As the existing monadic `I/O` framework [69] for Haskell compilers such as `ghc` does not allow actions on state components, a new monadic proposal [55] is presented, the main innovation of which is that it allows actions on state

[2]The monadic `I/O` mechanism has replaced the stream-based `I/O` one after Haskell 1.3 version onwards.

components. Moreover, it enables *state splitting*, a technique which assigns to each new thread a part of the state, a substate, to act upon. Such an extended monadic framework offers two additional advantages. It provides a modularised approach to state types, where each state type together with its primitive operations can be defined in its own module. Furthermore, the Brisk monadic framework improves some existing language features by making them deterministic, e.g. reading characters lazily from a file without unexpected effects. We also provide a lazy version of the standard Haskell function `writeFile` for writing characters lazily to a file. (*Chapters 4, 6*)

Additionally, this thesis investigates how to make deterministic concurrency more expressive by offering a form of *deterministic communications*. In the initial model of deterministic concurrency [51], threads communicate via shared values, including streams which they take as arguments or produce as results. However, concurrent program designs, particularly for `GUI`s, tend to use a freer style in which arbitrary networks of components, communicating via channels, are set up. This style encourages good modularity of components in an object-oriented style, but on the face of it, such unrestrained communication immediately re-introduces non-deterministic merging of message streams. In this thesis, such a freer style of communications is discussed, while preserving determinism. When employing a deterministic form of communications, input from several threads is merged in a deterministic way. To make it deterministic, the merging is based on timing information provided by timestamps attached to messages coming from different sources. These timestamps impose a linear order on all messages, allowing them to be merged in a consistent temporal order. Our system is described in terms of a network of processes which are connected via fixed uni-directional channels and communicate via messages. To each message, a timestamp is attached. A process is prevented from accepting a message until it has been determined that no earlier messages can arrive. This is achieved by the use of *synchronisation messages* which indicate which timestamp a message would have if it would been sent from a particular processor. (*Chapters 5, 7*)

Adding safe facilities for creating threads and channels leads to a system that corresponds to a deterministic subset of a concurrency model such as the $\pi$-calculus [83]. In this subset there may be internal non-deterministic evolution

but the observational output from each independent process (source of demand) is weakly bisimilar to a deterministic description acting over the relevant inputs.

### 1.1.2 Distribution in a Demand Driven Style

Concurrency is essential in distributed systems and since traditional concurrency models do not preserve referential transparency, distributed systems suffer from unwanted non-deterministic effects.

As an application of deterministic concurrency, we present a model of distribution that preserves referential transparency. Within the semantic framework that has been developed for deterministic concurrency, distribution is almost an orthogonal issue. Programs can be distributed without any change to their meaning, the only observable difference being the rate at which the overall system generates results. This is due to the fact that deterministic concurrency preserves both the declarative semantics and their operational equivalence.

The operational model of this distributed system uses a uniform access method in order to provide a distributed, transparent addressing scheme which ensures global knowledge of the current state of the system, where objects can be uniquely named. This scheme for unique naming of objects in the presence of distribution combined with:

- deterministic concurrency;
- explicit representation of demand;
- dynamic loading of graph fragments into the running environment;

provides the mechanisms to distribute reactive systems on several processors and offers a new model for computational mobility. As we have totally preserved referential transparency, any pattern of distribution has no effect on the denotational semantics of the system.

As a result, one main advantage of this distributed system is confidence in correctness of distributed programs. This is in contrast to conventional distributed programs, where the confidence in their correctness relies on both the correctness of the functionality of the program and also on the correctness of the communication conventions and timings of their distributed aspects. In the

distribution model presented here, it is guaranteed that confidence in correctness relies only on the correctness of the functionality of the program. Any pattern of distribution has no effect on the semantics of the functional program and therefore, given that the reliability of the underlying mechanism has been established, a programmer need only worry about functionality. This is due to the preservation of referential transparency through deterministic concurrency.

Moreover, this scheme allows data, code and computation to be moved in response to demand. This provides both a conceptual and operational model for mobile computation; program fragments can migrate from one processor to another and the location of computation can change at run-time. (*Chapter 10*)

## 1.2 The Tool - The Brisk Compiler

The Bristol Haskell compiler (Brisk) is a compiler for the Haskell language that acts as a tool for investigating the extension of purely functional languages towards concurrency and distribution without spoiling their semantic properties. The contribution of this thesis to the Brisk compiler is the Brisk Machine.

### 1.2.1 The Brisk Machine

Many different machine models have been proposed as intermediate forms in the compilation of functional languages, to act as the target of the front end of compilers and as a starting point for code generation. Their investment has mainly been to increase sequential performance and to provide industrial strength compilers, aiming to make functional languages industrially competitive and generalise the use of functional languages outside of academia. On the other hand, these performance design decisions make the resulting models complicated and restrict their flexibility and extensibility. (*Chapters 3, 8, 9*)

As a consequence, for the Brisk compiler we needed an abstract machine model based on a different philosophy. Thus, the Brisk machine has been designed to offer a machine model which is simple and flexible enough to support a number of different run-time execution models. In particular, design considerations have been made in order to support concurrency, dynamic loading and linking, distribution and computational mobility, debugging tools and logic pro-

gramming, in the form of the Escher [76] extensions to Haskell. On the other hand, traditional optimisations are orthogonal to this model and can be added without affecting any of the above mentioned extensions.

The simplicity of the Brisk machine relies on the fact that many issues concerning execution and optimisation are relegated to special built-in functions. To achieve this, the compiler's intermediate language called Brisk Kernel Language `BKL` is simpler than other intermediate languages used in existing compilers [100]. This leaves a lean and adaptable basic execution model which simplifies the run-time system and makes the execution of programs match closely the graph reduction model. Additionally, the simplicity of the Brisk machine model makes it *extensible* and *dynamic*.

- *Extensible* because it allows alternative approaches to evaluation and sharing to be chosen. As a result, extensions concerning optimisations, concurrency, distribution and logic programming have been added without any interference between them.

- Furthermore, since evaluation is hidden away in special built-in functions, every call (whether in an expression or on the right hand side of a local definition) must be saturated, which in turn requires that its function must represented as an evaluated function node. Thus, in the Brisk machine, function nodes (including global ones) are represented as heap nodes which contain references to each other. They follow the same layout as call nodes, being built from constructors as if they were data. This uniform representation of heap nodes makes this machine model *dynamic* and is the key issue that allows the Brisk compiler to offer dynamic loading, distribution and computational mobility, as:

  - new nodes can be created dynamically to cope with special situations;
  - functions can be loaded dynamically as newly created nodes in the heap;
  - newly compiled code can be dynamically loaded in the heap. This allows to mix compiled and interpretive bytecodes on a function-by-function basis;
  - code can be passed from one process to another.

The dynamic aspects of the Brisk machine support dynamic loading and computational mobility, as they allow code to be loaded dynamically into the running environment or communicated dynamically between machines. It also provides the basis for an integrated tool which allows execution to switch between compiled and interpreted code.

To complement the design of the Brisk machine, the run-time system of the Brisk compiler is also described. Special attention and focus is given in such run-time techniques that provide support for dynamic loading of newly compiled modules.

### 1.2.2 The Distributed Brisk Machine

The flexibility and extensibility of the Brisk machine are the key issues that allow it to be extended to support distribution. Adding distribution extensions to the Brisk run-time system has been facilitated by the design of the Brisk machine in the following aspects:

- extensions have been added in the form of built-in functions that hide the distribution details which operationally convey information across machine boundaries and provide an interface to the global heap;
- the uniform representation of expressions including functions, as heap nodes, not only provides support for dynamic loading of newly compiled modules but also enables heap fragments that represent data and code to be communicated dynamically between machines and loaded dynamically on another processor.

Adding distribution extensions to the Brisk machine offers a core which can be adapted to implement different policies to satisfy system constrains, such as load balancing or user constraints, such as mobility. It also shows that computational mobility in a distributed setting is facilitated by both deterministic concurrency and the design of the Brisk machine. (*Chapter 11*)

A simulation of distribution is presented which emulates distributed execution on a single processor based on the operational model described in Section 1.1.2 and Chapter 10 and demonstrates formaly the validity of distribution based on deterministic concurrency. This implementation is built into the Brisk

run-time system and consists of a collection of Brisk machines which exchange information via messages. Each machine consists of heap allocated objects together with an index table to keep track of references from other machines. The index table offers a uniform access method to heap objects and ensures global knowledge of the current state of the system, where objects can be uniquely named. Thus, the complete set of heaps can be thought of a forming a single global memory.

## 1.3 Contributions of the Thesis

The contributions of this thesis are as follows:

- The Brisk machine, (*Chapter 8 9*). The Brisk machine is an abstract machine for compiling Haskell, designed to offer a simple run-time system (*Chapter 9*). This is due to:

  - A minimal intermediate language, `BKL`;
  - Special built-in functions to handle evaluation, extensions, optimisations.

  The Brisk machine is also a dynamic model, as function nodes are represented as ordinary heap nodes. All heap nodes can be viewed as active objects responsible for their execution. This is important as it allows the following features:

  - Dynamic loading and linking. Current trends in compiler design favour dynamic loading and linking and most modern compilers are especially designed for this;
  - Compilation to interpretive bytecodes;
  - Compile code to be mixed freely with interpreted code. This provides the basis for building an integrated tool which allows execution to switch between compiled and interpreted code.

- Avoiding determinism in the presence of concurrency has always been a very challenging problem. Deterministic concurrency is a restricted form

of concurrency that does not allow non-determinism to appear at a user-level. Its conceptual model involves a number of independent threads that do not communicate with each other. This thesis integrates this deterministic concurrency model into the monadic `I/O` model of Haskell (*Chapter 5*). To achieve this, the basic concept is that of a state component (*sub-state*), upon which a thread can act.

- The need to support state components and actions upon them, led to the design of the Brisk monadic framework. This allows actions on state components, necessary for incorporating deterministic concurrency in Haskell (*Chapter 6*). Its main advantages are:
  - A modularised approach to state types;
  - Lazy Streams in Haskell, i.e. reading characters lazily from a file and writing characters lazily to a file without non-deterministic effects;
  - Actions on state components.
- Beyond acting upon distinct state components, threads need to communicate in a deterministic way. The next contribution of this thesis is a deterministic form of communications which allows for deterministic merging and offers deterministic time-merged channels (*Chapter 7*);
- As an application of deterministic concurrency, the operational model of a distributed system based on deterministic concurrency (*Chapter 10*) is presented. This offers a core upon which trustable distributed systems can be built. Its novel features are:
  - Unique Naming of Objects;
  - Explicit representation of demand;
  - Deterministic concurrency;
  - Uniform address space;
  - Infrastructure to support migration;
  - Variable granularity of mobility;
  - Potential support for long-lived data.

- Distribution extensions to the Brisk machine (*Chapter 11*). These integrate the operational model of distribution in the run-time system of Brisk:
  - provide special built-in functions that facilitate distribution;
  - extend the dynamic loading mechanism of the Brisk compiler to allow graph fragments to be communicated between machines in messages and loaded dynamically on the other end.

# Part I

# Foundations

# Chapter 2

# Concurrent Programming

The purpose of this chapter is twofold. On the one hand, it gives a short tutorial about concurrency, as this is traditionally studied. On the other hand, it discusses and questions some of its main issues, such as:

- concurrency is associated with parallelism and is used interchangeably in many contexts;
- concurrency is based on shared state;
- the need for a non-deterministic merge.

This is necessary in order to prove later that many aspects of concurrency are not related directly with the issue of concurrency itself but with the fact that concurrency has been mainly developed within the context of procedural languages. Many of the ideas of this chapter will be later invoked in order to set the scene for deterministic concurrency, a purely declarative form of concurrency.

## 2.1 Introduction

Concurrency is necessary in nearly all applications nowadays, from operating and interactive systems to animated and distributed applications. A concurrent program does several things at once and consists of a number of execution units (either processes or threads), that are supposed to be executed independently at the same time.

In early days of concurrency, concurrency was based only on processes which were statically defined from within the program for efficiency. Later, the evolution of software required a more flexible approach to concurrency, where concurrent systems were designed without the assumption on how many processes will be needed. This led to the inclusion of mechanisms for dynamic process creation in procedural languages. Still, processes were too monolithic to meet the requirements of today's software since several activities need to take place within each process. This gave rise to the notion of a thread as an execution unit within a process. Many programming languages and systems are designed or specifically adapted to support multi-threading. This helped support graphical user interfaces, distributed applications or Internet applications such as Web browsers. This chapter offers a brief overview of processes, threads and essential concurrency principles.

## 2.2 Processes

A *process* or *heavyweight process* is a basic unit of concurrency. Each process consists of a piece of program code that executes a sequence of statements and specifies a sequential flow of control. A process differs from a program because it cannot run on its own; it runs only within programs executing at its own rate, simultaneously with other processes.

A process also includes state to specify its current activity. For example, the state of a `Unix` process includes usually its location in the program specified by the program counter (`PC`), its procedure call context specified by the stack and the stack pointer register `SP`, its global variables specified by data in memory, files and devices specified by the `I/O` status, page tables, the processor status register `PSR` or other registers. Information about the state of a process together

with information about scheduling and process management is stored in a special descriptor, the process control block `PCB`.

### 2.2.1 The Life Cycle of a Process

The life cycle of a process includes the process creation, process scheduling and process termination phases.

**Process Creation** Current trends in concurrency favour dynamic process creation, where processes can create dynamically new ones by executing specific system calls. This is achieved by a specific forking mechanism which takes a piece of code as an argument and creates a new process to execute it. Each newly created process is called the *child process* whereas the creating process is called the *parent process*. For example, in `Unix`, processes are created by the system call `fork()`. This creates a child process as a copy of the address space of its parent. To allow processes to communicate more effectively, both parent and child processes run sharing the same address space after `fork()` has been executed. Although this is in contrast to the fact that processes should be independent of each other, it has been adopted as it has been considered the most effective way for them to communicate.

As concurrent systems became more and more important, many programming languages have been enriched by *fork* or *spawn* primitives that create multiple processes. Their common characteristic is that:

> *Processes are created so that they share the same address space and thus, they are not independent, despite the fact they give the impression that they are. This has been considered as the most effective way for processes to synchronise their activities. On the other hand, this is a major source of non-determinism as it authorises non-deterministic behaviour, e.g. a user may access two files simultaneously.*

**Process Scheduling** During the execution of concurrent programs, *fairness* must be guaranteed, i.e. all processes must have the chance to proceed. Fairness is ensured by a *scheduler* which defines which process will be executed next

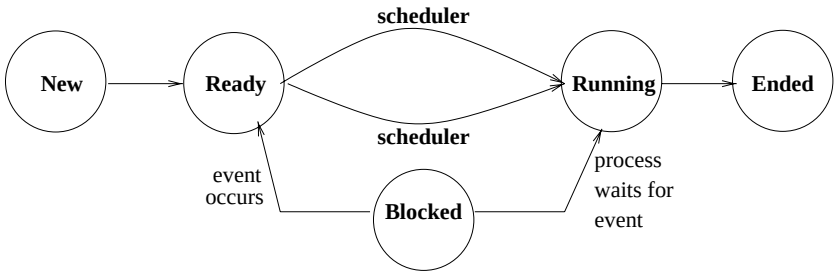



Figure 2.1: Scheduling Queues

among many candidate ones according to a particular scheduling algorithm. There is a number of scheduling algorithms which define a number of scheduling policies. A scheduling policy is *unconditionally fair* if it gives every process that is not delayed the chance to proceed. On a uni-processor machine, a round-robin scheduling policy is unconditionally fair, whereas on a multi-processor machine parallel execution is unconditionally fair. A scheduling policy is *fair* if it is unconditionally fair for processes that are not delayed and also every delayed process will eventually proceed.

To enable scheduling, each process has a *status* or *scheduling state* which is held in the process control block `PCB` and can either be: *new* for a newly created process, *ready* when a process is ready to execute, *running* when it is being executed, *blocked* when is suspended, i.e. waiting for `I/O` or *ended* when it has finished. There is one scheduling queue for each respective scheduling state, as can be seen in Figure 2.1. Processes with ready scheduling state belong to the ready queue, blocked processes belong to the device queue etc.

A process which is on the running queue runs for a certain period of time. During its execution, its scheduling state changes which causes it to move to another scheduling queue, e.g. it may be suspended waiting for some `I/O` and in this case it moves to the *device queue*. A running process also does not run for ever, it eventually moves to the ready queue, giving other processes also the chance to proceed. Transitions from ready to running queue happen via a scheduler, based on one of the following scheduling algorithms [114]:

- *FIFO* which serves processes as they enter the ready queue. Although it gives all processes the chance to proceed, i.e. it is unconditionally fair, it

cannot serve immediately urgent processes;

- *Priority Scheduling*, in which each process is associated a priority and the process with the highest priority is executed first. Although it can serve immediately urgent processes, it does not guarantee that low priority processes will be eventually served;

- *Round Robin*, which is a combination of `FIFO` with preemptive scheduling[1] (*timeslicing*) and is also unconditionally fair. The ready queue is treated as a circular queue and a *timeslice* is associated with each process. Each process runs for a timeslice. When its timeslice finishes, it is placed at the end of the ready queue.

When a process relinquishes control to another process, a *context switch* occurs, i.e. the process saves its state information allowing another process to take over. When a context switch occurs, the following steps are executed:

- The running process saves its local state information in the `PCB`;

- The `PCB` is added either to the ready or the blocked queue and the status of the process changes either to ready or to blocked respectively;

- The first `PCB` is extracted from the ready queue and the status of the corresponding process changes to running;

- Local state information is restored from the `PCB`.

Context switching decreases considerably the performance of programs and should be employed only when necessary.

**Process Termination** Processes may terminate voluntarily when they finish executing using the `exit` system call. In this case, they relinquish their resources to the operating system and possibly data to their parent. Alternatively, processes may be forced to terminate by their parent using the system call `abort`, either because the child process has exceeded the usage of resources or it is not required any more by the parent or the parent itself needs to exit.

[1] *Preemptive* scheduling is the scheduling which takes place either when a process switches from the running state to the ready state or from the waiting state to the ready state [114]. When preemptive scheduling occurs, the process releases its allocated `CPU`.

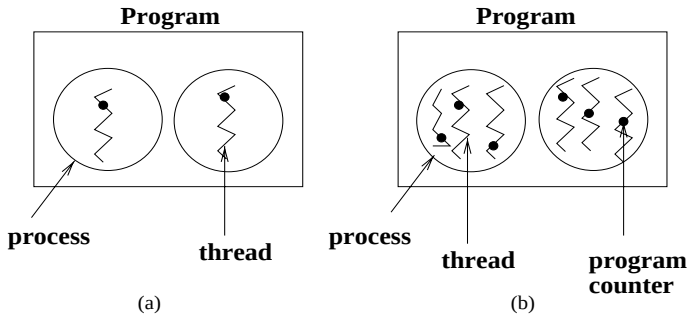



Figure 2.2: (a) Single Threaded Processes. (b) Multi Threaded Processes.

## 2.3 Threads

A *thread* or *lightweight process* is an independent execution sequence within a process. Threads do not have their own private state as processes but share the state and global variables of a process together with other threads. Thus, threads offer a more flexible unit of concurrency because they can take more advantage of both the resources and the environment of a particular program and enable them to be shared and accessed concurrently. On the other hand, threads are more dangerous because they can interfere easier with each other, e.g. writing on one another's stack, so synchronisation techniques, which will be discussed in Section 2.4 are even more indispensable.

The use of threads emerged out of the need for better `CPU` utilisation [114], since multiple threads consume fewer operating system resources than multiple processes and provide inexpensive context switching. Thus, modern operating systems, such as Solaris, are multi-threaded. Their use has also been generalised beyond operating system design and most concurrent applications are multi-threaded; graphical user interfaces where multiple windows can be created, distributed applications where application programs communicate concurrently sharing the network that interconnects them or Internet applications such as Web browsers where threads create multiple connections to servers and display items for the user while other activities are going on, to mention some of them.

To illustrate further the concept of a thread, a `Unix` thread will be taken as an example. Each thread has its own program counter register `PC`, stack

and stack pointer register SP. Threads may share freely between them data in memory, I/O status, memory, page tables, processor status register PSR, files and devices. Since threads have only their own register set, they enable cheaper context switching because it involves only a register-set switch and not any memory management or other activities. Usually threads programming is done in a conventional language such as C enriched with a threads library. Such examples are the C threads package developed for the Mach operating system and the SunOS Lightweight Processes (LWP) package. The need for standardisation led in two main thread interfaces for Unix, the IEEE Portable Operating System Interface standard (POSIX) also called the Pthreads library and the Unix International threads standard (UI) [65]. Also, many procedural (Java [17, 41]) and declarative languages (ML, Haskell) have been specially designed or adapted to provide direct support for threads.

The life cycle of a thread is similar to that of a process. It consists of the creation, scheduling and termination phases. As with processes, the parent thread continues running after the creation of the new thread. Threads have also similar scheduling states as processes and move during execution between scheduling queues according to a particular scheduling algorithm or when they block, e.g. waiting for I/O.

To be suitable for multi-threading, many programming languages have had special primitives added for creating and handling threads. Similarly to dynamic process creation, primitives for dynamic thread creation immediately introduce non-determinism and synchronisation techniques are required to prevent threads from interfering between them.

## 2.4 Synchronisation

Concurrency is heavily based on shared state since many facilities need to be shared by several processes or threads. When several processes compete for the same resource, non-deterministic effects arise.

Thus, both processes and threads[2] need to synchronise among them in order to cooperate effectively when sharing resources or exchanging information. The

[2]Since synchronisation techniques apply equally to both processes and threads, both words will be used interchangeably within the context of synchronisation and communication.

most important synchronisation [9, 10, 4] methods are:

- *mutual exclusion*, which guarantees that at most one process at a time can proceed in the critical section, a code area that can lead to interference;
- *condition synchronisation*, in which the execution of a process is delayed until the program state satisfies some condition.

### 2.4.1 The Critical Section Problem

A *critical section* is a code segment in a process that accesses shared resources which may be accessed also by other processes. Only one process must access its critical section at a time. The critical section problem consists in inventing a protocol so that processes can use it to cooperate safely when accessing such shared resources. A formal representation of this problem [4] assumes `n` processes that repeatedly execute a critical section of code, then a non-critical section. The critical section is preceded by an *entry protocol* and followed by an *exit protocol*.

```
Process Pi, 1 ≤ i ≤ n
while (1) {
   entry protocol;
   critical_section;
   exit protocol;
   non_critical_section;
}
```

A solution to the critical section problem must also satisfy some properties. A property of a program is an attribute that is true in every possible history of it. Properties are usually divided into two important categories, *safety* and *liveness* ones [4]. A safety property ensures that a program never enters a bad state and none of its variables has an undesirable value. A liveness property asserts that a program eventually enters a good state, i.e. all its variables have desirable values. In sequential programs, an example of a safety property is partial correctness, i.e. the program terminates and an example of a liveness property is that the result of a program is correct.

A solution to the critical section problem must satisfy the following safety and liveness properties:

- *mutual exclusion*, i.e. the critical section is accessed only by one process at a time so that non-deterministic effects which arise when two or more processes access simultaneously the same resources are excluded. This is the most important prerequisite for the critical section problem and several synchronisation techniques which ensure mutual exclusion are discussed below;
- *absence of deadlock*, i.e. if two or more processes are trying to enter the same critical section, at least one will succeed;
- *absence of unnecessary delay*, i.e. if a process is trying to enter a critical section and the other processes are executing their non-critical sections or have terminated executing in their critical section, the first process is not prevented from entering this critical section.
- *eventual entry*, i.e. a process that requires access to a critical section will eventually succeed.

Mutual exclusion and absence of deadlock are safety properties, i.e. ensure that a concurrent program never enters a bad state, i.e. none of its variables has an undesirable value. Eventual entry and absence of unnecessary delay are liveness properties, i.e. guarantee that a concurrent program eventually enters a good state, i.e. all its variables have desirable values.

### 2.4.2 Mutual exclusion

As already explained, when concurrent processes need to share main storage, mutual exclusion is usually employed to ensure that resources are not accessed simultaneously. Although mutual exclusion is traditionally voluntary, it is a matter of good practice and all concurrent applications rely on this. The next sections illustrate the most important techniques used to guarantee mutual exclusion.

**Locks** The most straightforward way to approach mutual exclusion is via simple boolean synchronisation variables, or *locks* that indicate whether a process is in the critical section or not. The advantage of this approach it that it can be implemented directly in hardware, software and operating systems. Its disadvantage is that most solutions are based on *busy waiting*, i.e. a process needs to keep checking when the critical section becomes available.

The hardware (coarse-grained) approach requires one lock variable and a special hardware instruction such as `Test-and-Set` or `Fetch-and-Add`, available in most multiprocessors. Although this approach works well on multiprocessors, its disadvantage is that is both machine dependent and requires a strongly fair scheduling policy which is expensive to implement. Also for architectures that do not provide such an instruction, Lamport [68] invented an algorithm that offers its functionality.

There is a number of solutions for the software approach (fine-grained), the most prominent of them being Peterson's `tie-breaker` algorithm. This requires only unconditionally fair scheduling to satisfy the eventual entry requirement, it does not need special hardware instructions and it can be generalised for `n` processes. On the other hand, it is considerably more complicated, i.e. three locks are required. An extended overview of fine-grained and coarse-grained solutions can be found in Andrews [4].

The operating system approach requires only one lock variable, disables interrupts before entering the critical section and enables them afterwards. Processes that wait to get the lock are stored in a queue. This approach avoids busy waiting but does not work well on multiprocessors.

The Java programming language offered a novel approach to mutual exclusion where a critical section is a method in an object (class) which is tagged by the keyword *synchronized*. Whenever control enters a synchronised method, the thread that called the method locks the object and prevents other threads from accessing it.

All synchronisation protocols based on simple synchronisation variables tend to be rather complex and unsuitable for multiprogramming because of extra complexity, since they don't separate ordinary variables from communication variables. Furthermore, most of them rely on busy waiting which decreases

performance. This led to the invention of alternative approaches to mutual exclusion such as semaphores and monitors which are described in the next paragraphs.

**Semaphores** Dijkstra suggested semaphores as a more structured approach to implement mutual exclusion. A *semaphore* is a variable which has a non-negative integer value together with two operations, `wait(s)` and `signal(s)`:

```
wait(s): if s > 0 then s = s-1 else wait until s > 0
signal(s): s = s+1
```

where `wait(s)` blocks the process that executes it until the value of `s` is positive. If the value of the semaphore ranges between `1` and `0` the semaphore is called *binary*, otherwise it is called *general*. When using semaphores, the critical section problem is formalised as follows.

```
Semaphore s := 1;
Process Pi, 1 ≤ i ≤ n
while (1) {
   wait(s);
   critical_section;
   signal(s);
   non_critical_section;
}
```

Semaphores can be implemented in many ways, with or without busy waiting. They can also solve more sophisticated forms of mutual exclusion, like the *dining philosophers problem* posed by Dijkstra or the *readers and writers* problem.

**Monitors** *Monitors* are a more structured way to achieve synchronisation than semaphores and their use can be enforced by compilers. A monitor is an abstract data type which represents resources together with a set of operations as the only way by which the abstract type can be accessed and manipulated. A monitor contains also variables that store the state of a resource and processes can only access these variables by calling one of the monitor procedures. Mutual exclusion is provided implicitly by ensuring that at most one process at

a time may be executing within any procedure. Monitors provide also special facilities called *condition variables* that enable another form of synchronisation called condition synchronisation which is discussed in the next section 2.4.3. The only operations on a condition variable are `wait` and `signal`. A variable `c` delays a process that cannot safely continue executing by executing `wait(c)` until a boolean condition is satisfied. The value of `c` is a queue of delayed processes which is not visible to the programmer. When the condition is satisfied, `signal(c)` removes the process from the delay queue and activates it again.

Monitors have been incorporated in the Java programming language [17, 41]. As already mentioned, a critical section is a method in an object (class) which is tagged by the keyword *synchronized*. Whenever control enters a synchronised method, the thread that called the method locks the object. If other threads attempt to call this method, they are placed in a queue. The `Thread` class provides the methods: *wait* for placing a thread at a waiting queue until the object is unlocked and *notifyAll* for notifying any threads at the waiting queue that the object is unlocked.

### 2.4.3 Condition Synchronisation

A whole class of problems in concurrency require the delay of the execution of a process until the program state satisfies some condition. The most representative example of this class of problems is the *producer-consumer* problem. This consists of two processes, a producer that produces items and a consumer that consumes them. It encapsulates the synchronisation problem that arises when the producer needs to store items until the consumer is ready to consume them and also when the consumer tries to consume items when they have not yet been produced. Examples of the producer-consumer problem are a print program which produces characters that are consumed by the printer driver, a compiler which produces assembly code which is consumed by the assembler or an assembler which produces object modules that are consumed by the loader [114]. One solution to the producer-consumer problem is by means of synchronous communication arranging a rendezvous between consumer and producer as it is discussed in Section 2.5.1. Alternatively, producer and consumer may cooperate asynchronously in a shared memory environment by sharing a

common *buffer*. A buffer can be either *bounded* or *unbounded*. An unbounded buffer does not restrict the size of the buffer, whereas a bounded buffer provides a fixed buffer size. In fact, a buffer implementation requires both mutual exclusion and condition synchronisation. Mutual exclusion ensures that the producer and the consumer do not access the buffer simultaneously. Condition synchronisation for an unbounded buffer ensures that the consumer waits if the buffer is empty, whereas for a bounded buffer ensures that the consumer waits if the buffer is empty and also that the producer waits if the buffer is full.

### 2.4.4 Important Synchronisation Problems

**Dining Philosophers** This problem encapsulates situations that arise when a process requires simultaneous access to many resources. Five philosophers are seated around a circular table, thinking and eventually eating out of a large plate of spaghetti placed in the middle that acts as the critical section. There are only five forks and each philosopher needs two forks to eat. Each philosopher also can use only the forks to the immediate left and right. A philosopher gets hungry, tries to pick up two forks and enters the critical section and eats. When finished, a philosopher puts down the forks and starts thinking again.

**Readers and Writers** In the readers and writers problem there are two kinds of processes that share a database or any other shared data object, readers that examine the object and writers that both examine and update the object. Nondeterministic effects arise when a writer and any other process (a reader or a writer) access the object simultaneously. Any solution of this problem must guarantee that any number of readers can access simultaneously the database but only one writer can access it at a time.

## 2.5 Communication

Processes need also to communicate by passing data between them. There are two ways for processes to communicate, via *shared variables*, or via *message passing*.

Communication by shared variables is similar to the synchronisation meth-

ods that have been explored so far. It is based on the assumption that processes share common memory and communicate via shared variables which are stored within it. When processes communicate via shared variables one process writes into a variable that is read by another. This form of communication is efficient on uni-processors or on shared memory multiprocessors but it is inefficient in case of virtual memory or distributed systems. Additionally, it guarantees no protection, since any process can access another's memory.

When communicating by message passing, processes are assumed to share a communication network and exchange data in messages via `send` and `receive` primitives. This offers a better form of communication for distributed computer architectures and distributed memory languages. This form of communication ensures also implicitly synchronisation, since a message cannot by received before being sent. The simplicity and expressiveness of the message passing model makes it popular. Current trends in concurrent applications favour communications by message passing not only in multi-processors but also in a uniprocessor or between threads. For example, in graphical user interface design, graphics libraries are often presented in terms of a collection of processes which communicate asynchronously with each other by sending messages via channels. This model offers good modularity of components in an object-oriented style because it allows to separate the event network (input) from the graphics network (output). Graphics is organised into hierarchical windows, whereas event handling is organised in terms of an arbitrary network of processes which communicate asynchronously by sending messages to each other via channels and an arbitrary communication network links them. The communications network can be made arbitrary by naming communication channels.

Communication by message passing can be either *synchronous* or *asynchronous* depending on whether the sending and receiving processes synchronise or not.

### 2.5.1 Synchronous Message Passing

In synchronous message passing, the sender process delays until the receiving process is ready to receive the message. This implies that sender and receiver need to synchronise at every communication point. However, messages do not

have to be buffered. If the sender proceeds, this means that the message has indeed been delivered.

Synchronous message passing has been based on Hoare's *communicating sequential processes* (CSP) [49]. In CSP, sender and receiver processes communicate by naming each other. An input statement has the form $P?x$, where process $P$ receives message $x$ and an output statement $P!x$ where process $P$ sends message $x$. A process $P$ that sends process $Q$ a message $Q!x$ must wait until the process $Q$ executes $P?x$. Both processes $P$ and $Q$ are synchronised while communication takes place. This means that the evaluation of an expression delays until the communication at the other end is established. Synchronous communication results in a distributed assignment statement, where the evaluation of an expression happens at the sender process and the result is assigned to a variable on the receiving process.

Languages with synchronous communication are Occam [75] and Ada [91]. Occam has been based directly on the ideas of CSP. Its difference with CSP is that instead of having sender and receiver name each other, processes communicate via unidirectional channels, where a *channel* provides a direct link between processes. A process names a channel, i.e. sends a message to a channel and the receiving process obtains the message using the same channel.

Ada provides a synchronous communication model with bidirectional channels based on rendezvous. A rendezvous is a communication mechanism that combines aspects of monitors and synchronous message passing. Similarly to monitors, a module or process exports operations to be called by the other end. Similarly to synchronous message passing, the calling process delays until the rendezvous has been serviced and the results have been returned. This offers the advantage that an operation is a two way communication channel.

**Selective Communication** The most important theoretical contribution of CSP has been the idea of *selective communication* or *guarded communication*. This offers a theoretical framework upon which programming languages and systems have been based to provide facilities that handle situations which arise when a process has the option to choose non-deterministically between many alternative ones to communicate. Such facilities enable programs that wait for

input from different sources to respond to whichever item appear first from any of these sources. In the literature, this is described as the *disjunctive waiting* problem and the non-determinism that results out of this situation is referred also as *local angelic non-determinism* [49, 117]. Selective communication has been based on Dijkstra's [30] guarded commands. Guarded commands are expressions prefixed by boolean expressions called *guards* and are executed only when guards succeed. In `CSP` the notion of guarded command has been extended so that guards prefix both expressions and input operations. If the guard fails, then the command is not executed. If the guard succeeds and the expression it prefixes is not an input operation, then the command succeeds, otherwise the guard blocks until an output command corresponding to this input command becomes ready, i.e. communication from another process has been established. At a second level, `CSP` defines alternative commands. An *alternative command* consists of a number of guarded commands. If all guards of an alternative command fail, the alternative command fails, if no guards succeed and some guards are blocked, execution delays until the first input statement finds a matching output one, otherwise one of the successful guarded commands is selected non-deterministically and executed. Non-determinism means that a process that accepts events from multiple processes is not able to tell which of the processes a particular event came from and so it might not know to which process to send the reply back. Finally, at a third level, a *repetitive command* specifies as many iterations as possible of an alternative command. When all guards fail, the repetitive command terminates. If all guards block, execution delays until a guard succeeds or terminates. If some guards succeed, the alternative command is executed once by non-deterministically selecting one of the commands with successful guards and then the repetitive command is executed again. This is equivalent to waiting for another input guard to succeed while some background processing takes place. A repetitive command at its full generality is usually used for theoretical purposes and it is rather expensive to implement.

Most programming languages provide a primitive that enables processes to communicate with multiple inputs when the order of communication is unknown. Occam provides the `ALT` construct which offers the functionality of a

repetitive construct, whereas `Ada` [39, 10] provides a `select` construct which is simpler, since only one process can succeed at a time. As an alternative to selective communication, `Ada` implements polling where each process needs to check constantly whether its guard succeeds. This results in busy waiting and wastes resources but on the other hand it is easier to implement.

Operating systems such as `Unix` also provide a `select` system call which allows to implement concurrent `I/O` by handling input items or events coming from different sources. This is passed the identifiers of a number of input channels and returns the first item to arrive on any of the channels. As an example, the `select` system call allows a program to read characters from both a `TCP` connection and the keyboard. If the program blocks waiting data from the `TCP` connection and the user types a command while the program is blocked, the `select` call informs the program which source of input became available and the program proceeds further. In graphical user interfaces or other stream based programming, a merge facility is provided which, in contrast to a `select` system call which is internal to programs, it operates externally to programs in the sense that programs access a single queue of events merged from different sources.

### 2.5.2 Asynchronous Message Passing

In *asynchronous message passing* both sender and receiver processes proceed independently and communication channels are unbounded queues of messages. A sender process continues executing after sending a message and a message might be received arbitrarily long after it has been sent. A process appends a message to the end of a channel's queue by executing a form of a `send` statement. Since the queue is unbounded, `send` does not block. A process receives a message from a channel by executing a `receive` statement. The execution of `receive` might delay the receiver until the channel is non-empty; then the message at the front of the channel is removed and stored in values local to the receiver. Examples of languages based on asynchronous message passing are *Actors* [2] or PICT [108] based on the $\pi$-calculus [83].

Similarly to synchronous message passing, selective communication is required for a process to select between many alternative ones to communicate.

## 2.6 Concurrency Revisited

In the previous sections, the concept of concurrency has been presented together with its most important issues and examples, as it is usually studied in the literature. In this section, we attempt to consider concurrency from a different point of view and highlight some of its issues which will set the scene for adapting concurrency to a purely declarative style.

The first important issue to discuss is that the theory of concurrency does not take into account the purpose of concurrent processes. Generally in concurrent systems, processes may either cooperate towards a common goal as in the case of *parallelism*, or serve distinct requirements as in the case of *explicit concurrency*. Both parallelism and concurrency have been treated inseparably and studied together in the literature because:

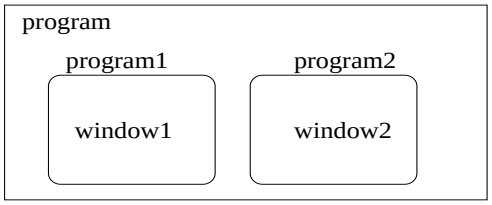
- concurrency offers an abstract framework free from implementation details into which parallelism can also be studied;
- concurrency requires some measure of parallelism in its implementation;
- concurrency + parallelism = high performance.

Thus, from the implementation point of view both issues are based on the following common characteristics:

- The use of processes or threads is required to carry out distinct activities;
- A scheduling policy is required to ensure fairness;
- Synchronisation and communication are required to enable a deadlock free coordination of distinct activities and data exchange.

For the purposes of this thesis, a more formal distinction between parallelism and concurrency needs to be drawn.

*Parallelism* is taken to mean the use of multiple processes in the internal implementation of a program in a way that does not affect either its semantics or its externally visible behaviour, except in terms of efficiency. It only affects the rate at which results appear. The expressive power of programs though, is not increased beyond sequential execution; although many activities may occur




(a) The Two Window Example


(b) Non-deterministic Merge

Figure 2.3: Inherent Concurrency

at the same time, they are coordinated in such a way that their results are the same as if a program was executed sequentially.

On the other hand, *explicit concurrency* or *inherent concurrency* denotes the ability of a program to do several things at once in an asynchronised way, so that its externally visible behaviour becomes more expressive. A simple test whether a language has concurrency or not is whether it can solve the problem of Figure 2.3. Take two existing interactive processes, such as two processes running in two separate subwindows of the program's main window and combine them into a single program. The program should run both processes at the same time with no interference between them. If a mouse click or key press in one subwindow engages a long computation, the program should be able to interact with the process in the other subwindow. Inherent concurrency is indispensable for implementing a whole class of programs often described under the term *systems programming*. This class includes operating systems, reactive systems, graphical user interfaces or other low-level, communications-based software. Furthermore, concurrency is indispensable in *distributed programming* to facilitate interaction with remote programs. For the rest of this thesis, the term concurrency will be used to indicate only explicit, concurrent `I/O` performing processes or threads.

Another important point to mention is that concurrency is associated tightly with non-determinism. One main source of non-determinism is thread creation.

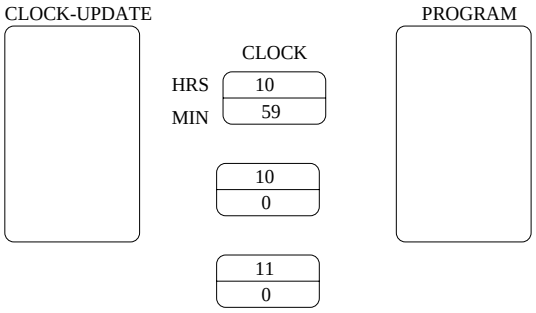



Figure 2.4: Mutual Exclusion Does Not Ban Non-determinism

In all languages, new threads are created using a *fork* or *spawn* primitive to create threads. New threads are created so that they share the same address space and thus they can easily communicate. On the other hand, this immediately introduces adverse non-deterministic effects. Several mutual exclusion techniques that have been described in Section 2.4 are employed to reduce these effects at least to some extent.

The following example [3], see also Figure 2.4, illustrates that mutual exclusion is a semi-safe approach. It consists of three processes, a clock process which acts as a critical section, a process that reads information from the clock and a process that updates the clock. The process that updates the clock can be thought of as having two synchronised methods, one for updating the hours and one for updating the minutes. Suppose the process that updates the clock enters the critical section and the hour is `10.59`a.m. Its minutes method changes the minutes slot to `0`, but then the process has to leave the critical section before the other method change the hours slot into `11`, since it is the other process's turn to enter the critical section. The time information the program gets at this particular moment will not be correct. Eventually the process will re-enter the critical section and will update the clock so that the hour is set to `11.00`. This example indicates that mutual exclusion is not enough to maintain the consistency of data. It also proves that even when applying mutual exclusion, the risk of creating unwanted non-deterministic effects is still present and can appear as undesired behaviour under certain, usually rare, sequence of events. The fact that these situations are rare has been not enough to outweigh the

[3] This example is given by David May

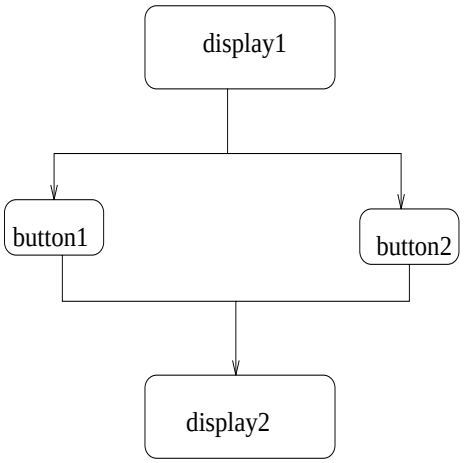



Figure 2.5: A Counterexample

gains of mutual exclusion.

The other main source of non-determinism in concurrent systems is when processes or threads communicate with each other via message passing. As already explained, current trends in inherently concurrent systems favour communication by message passing even on a uniprocessor. This requires selective communication which is also a major source of non-determinism. As already explained, selective communication allows a program to handle input items or events coming from multiple sources; when a program waits for input from different sources, it needs to respond to whichever item appear first from any of these. In the two window example of the Figure 2.3, key presses or mouse clicks from either window are assumed to form either two separate input streams or one input stream which is dispatched. Suppose the program is idle and that you press a key in one of the windows. The program needs a mechanism to enable it to respond to an item appearing on either of the two input streams, whichever appears first. Since the arrival of items depends on accidental timings of their production and delivery, the most obvious way to handle this situation has always been to non-deterministically merge items coming from these inputs.

The following example attempts to prove why a non-deterministic merge offers an unsatisfactory solution. It captures the problem caused by merging streams of information in concurrent systems. This simple example, see also

Figure 2.5, consists of two displays and two buttons. The user can type numbers at `display1`. The first button (`button1`) is associated with a complex function which takes as input the value from the first display (`display1`), performs a calculation and displays the result in the other display (`display2`). The second button (`button2`), when pressed, copies the value from the first display (`display1`) into the second display (`display2`). Suppose a user presses `button1` and immediately after `button2`. It is obvious that the user expects to see the results appearing in the order he pressed the buttons. (Un)surprisingly, the results will come out in reverse order, because they are displayed according to how fast they are produced. The result of pressing `button2` will be immediately delivered since it involves a simple copy, whereas the result of pressing `button1` will delay to appear as it involves a more complex computation the result of which will take longer to be available. This is because most implementations of such systems depend on some variation of a non-deterministic merge, which delivers the first item that is produced. But merges based on availability fail here and the results of the two button presses appear in reverse order, according to how fast they are produced. Of course, a user would expect to see results appearing in the order the user pressed the buttons.

Although a non-deterministic merge seems to be the only option to the disjunctive waiting problem, it is clear that it is an unsatisfactory solution. It fosters non-deterministic effects in inherently concurrent systems which usually manifests as undesired behaviour under certain accidental circumstances of timing, called race conditions. The fact that these bad timing conditions are normally rare, e.g. only appearing when a computer is under heavy load, makes them more difficult to detect and track down. On the other hand, due to the fact they are rare, they have been taken for granted and no further research has been undertaken to eliminate them. The need for a non-deterministic merge has been accepted as a necessary evil and the damaging effects of employing it have largely been neglected. One reason for this is that the implementation of concurrency has always been based on imperative languages and to some extent it has been influenced by their philosophy. Imperative languages have complicated semantics due to the uncontrolled update of memory locations which introduces complications and consistency problems when accessing updatable

shared values even in a sequential setting. The additional damages of introducing non-determinism to the already complicated semantics of imperative languages could be certainly neglected and has indeed been. Non-determinism has only been eliminated to the extent that the behaviour of programs is acceptable enough. Moreover, since the correctness of programs usually relies on speed, this kind of bugs always survive testing. It turns out though, that in a distributed setting, all these bugs which can be neglected on a single processor, become more serious under of the effect of network load, just when the user least wants them. As a consequence, today's software is riddled with these programming bugs. For example, even the common convention of double-clicking a mouse button, where two clicks which are close enough together in time are treated as having a special meaning, does not work over a remote connection to a computer if the times of the clicks are measured at the remote end.

> *Thus, non-deterministic problems associated with concurrency conspire to make distributed systems even more difficult to debug, adapt and maintain.*

# Chapter 3

# Functional Language Implementation

## 3.1 Lazy Evaluation

A functional program is evaluated by replacing some subexpression by a simpler, equivalent one. This is called *reduction* and each simplification step, a *reduction step*. The subexpression which is being simplified is called a *reducible expression* or *redex*. Reduction finishes when there are no more redexes. In this case the expression is said to be in *normal form*. As an example, consider the function definitions:

```
double n = n+n
square n = n*n
```

The expression `double (square (1+3))` can be reduced selecting always the *innermost* expression, as follows:

```
double (square (1+3)) ->
double (square 4) ->
double (4*4) ->
double 16 ->
16+16 ->
32
```

An innermost reduction strategy corresponds to the situation where all the arguments of a function get evaluated before calling it.

Alternatively, an expression can be reduced selecting always the *outermost* expression to reduce:

```
double (square (1+3)) ->
(square (1+3))+(square (1+3)) ->
((1+3)*(1+3))+(square (1+3)) ->
(4*4)+((1+3)*(1+3)) ->
16+(4*4) ->
16+16 ->
32
```

Although an outermost reduction strategy requires more steps to evaluate an expression, it is guaranteed by the Church-Rosser theorems [120] of the lambda calculus that both evaluation strategies yield the same answer if there exists one. A particular outermost evaluation strategy, the *leftmost outermost* evaluation strategy is called *normal order* reduction and corresponds to calling a function with unevaluated arguments. This reduction strategy has an important termination property:

> *Normal order reduction is guaranteed to terminate when there exists a normal form.*

To explain this, consider the following Haskell program:

```
f n = (n != 0) && ⊥
```

Evaluating `f 0` will only succeed when a leftmost outermost evaluation strategy is used.

```
f 0 = (0 != 0) && ⊥ ->
f 0 = False && ⊥ ->
f 0 = False
```

With leftmost outermost evaluation strategy, evaluation of expressions is delayed until they are needed. Moreover, if the value is never needed, it is not evaluated at all. This is also called *lazy evaluation* or *demand driven* evaluation.

On the other hand, an innermost evaluation strategy, also called *strict* or *eager* evaluation, requires the evaluation of all arguments before calling a function, but would fail to produce a result in the above case. A function is called strict or *eager* (`f` $\perp$ `=` $\perp$), if it needs the value of its argument. In other words, if the evaluation of the argument will not terminate, then the application of `f` to the argument will fail to terminate. *Strictness analysis* is a technique that detects for each function the values of which arguments are needed in order to compute the result. These arguments can be safely evaluated before calling the function without destroying the call by need semantics. Thus, strictness analysis improves the efficiency of programs as it allows some arguments to be evaluated in advance.

Lazy evaluation is often referred as *call by need*, whereas strict evaluation as *call by value*. Functional languages can be either *strict* or *lazy*. `ML` is a strict functional language whereas Haskell is a purely functional language.

One main advantage of lazy evaluation strategy is that it enables the manipulation of infinite data structures without causing the program to get stuck in an infinite loop as in procedural languages, since each element is generated according to demand. This offers simple and smart solutions to problems. For example, it is very simple and natural to define the output from a random number generator as an infinite list of numbers [50]. Only those numbers which are actually needed to produce the results get evaluated.

```
random s n = [(r `div` 1000) `mod` n | r <- rand s]
rand s =
   let s1 = s * 1103515245 + 12345
       s2 = if s1 >= 0 then s1 else s1 + 2^31 in
   s2 : rand s2
```

The `rand` function yields the same random sequence as the standard Unix `rand` procedure (at least on a 32-bit machine with no overflow detection). Function `random s n` returns an infinite list of (pseudo) random integers in the range 0 to (n-1). The number `n` should be between 2 and about 1000000. The function `random` can be called only once in any one run of a program and then the numbers in the result list can be used as needed. The integer `s` is an arbitrary

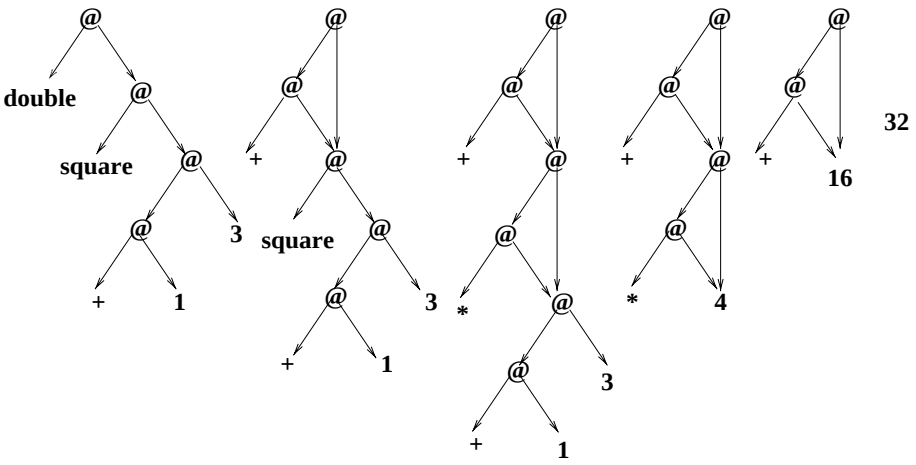



Figure 3.1: Graph Reduction

seed which can be used to give a different random number sequence on each run of a program.

## 3.2 Implementation of Functional Languages

### 3.2.1 Graph Reduction

In the previous section, normal order reduction has been illustrated. This forms the basis of lazy evaluation. Although it has important termination properties, it is highly inefficient due to the multiple evaluation of the same subexpression. *Graph reduction* is an optimised form of reduction, proposed by Wadsworth, which overcame this inefficiency and turned reduction into a practical technique [134]. It consists in introducing sharing of subexpressions and thus of their results, so that multiple evaluation of subexpressions is avoided. This is equivalent to representing expressions as graphs, where subexpressions consist the nodes of the graph, linked together with pointers. In real implementations, the graph is stored in a heap and each node corresponds to a heap cell. In each evaluation step, a node is overwritten by an equivalent one. When a subexpression (node) is evaluated, its value can be shared by all nodes that point to it. In this way, multiple evaluation of the same subexpression is avoided. For example, the expression `double (square (1+3))` can now be reduced as follows:

```
double (square (1+3)) ->
m+m where m = (square (1+3)) ->
m+m where m = n*n, n = 1+3 ->
m+m where m = n*n, n = 4 ->
m+m where m = 16 -> 32
```

The graph reduction of this expression can be seen in Figure 3.1.

The next sections describe how graph reduction is performed in modern compilers.

### 3.2.2 Graph Reduction in Modern Compilers

Functional languages are studied by relating them to the lambda calculus [8], a theoretical tool with minimal syntax often described in the bibliography as the first functional language. This provides a "lowest common denominator" for functional languages in which the theory of functions, their semantics and evaluation is carried out [50]. For the purposes of graph reduction, a functional language is translated into an enriched form of lambda calculus. An expression in the enriched lambda calculus can be either a variable, an application, a built-in function, a lambda abstraction, a let or letrec expression, a case expression or a pattern matching lambda abstraction. Built-ins can be arithmetic operations, constants, logical functions, logical operations, conditional functions and data constructors. For a detailed description, see also [98].

After the translation into lambda calculus, the next redex needs to be selected. Laziness requires that the next redex be selected using a leftmost outermost strategy. This is achieved by starting at the root and following the left branch of each application node, until a built-in or a lambda abstraction is encountered. This left-branching chain of application nodes is called a *spine* and this process is called *unwinding* the spine [98]. During the unwinding, pointers to the addresses of nodes encountered on the way down are saved in a stack for later use. If the redex is a built-in, all the arguments whose values are needed are first evaluated, then the built-in is executed and the root of the redex is overwritten with the result. If the redex is a lambda abstraction, reduction consists in constructing a new instance of the body of the lambda abstraction, where arguments are substituted for the free occurrences of the formal parame-

ter, i.e. a *beta* reduction is performed. The arguments are not copied; pointers substitute the argument for the formal parameter and the lambda expression becomes shared.

The efficiency of graph reduction depends heavily on how the function body is instantiated with the actual parameters. Early implementations such as the `SECD` machine [109] allowed code to access the values of free variables by using an environment which contained values for each free variable. Since this approach was too inefficient, *supercombinator graph reduction* has been undertaken. This is based on the idea of transforming a functional program into a supercombinator program, i.e. a set of functions without free variables or other lambda abstractions. This transformation is called *lambda lifting*. A more efficient variation of this technique [127] has been to transform a program into a fixed set of supercombinators called *SK combinators*, rather than having an arbitrary number of supercombinators generated from the program. This offered the advantage of having a fixed set of rules which can be implemented directly in hardware and gave rise to the notion of instructions of an abstract machine. The next sections describe the most important abstract machines.

### 3.2.3 The G-machine

The first abstract machine [59], [6] based on supercombinator reduction is the `G-machine`. Supercombinators are translated into a sequence of machine instructions, which form the abstract machine code or `g-code`, for efficiency. The abstract machine instructions, when executed, construct an instance of the supercombinator body. They also provide the operational semantics of programs. Reduction proceeds by repeatedly unwinding the spine and updating until a normal form is reached. In the `G-machine` nodes have a fixed size representation and its state has four components: the heap, the argument stack, the code and the dump. The heap holds the nodes representing the program graph. The stack is used during unwinding for "remembering" the nodes encountered on the way down the spine. The dump is a stack of pairs, the first component is a code sequence and the second one a stack. It stores mainly suspended calls of built-ins with unevaluated arguments until their arguments are evaluated.

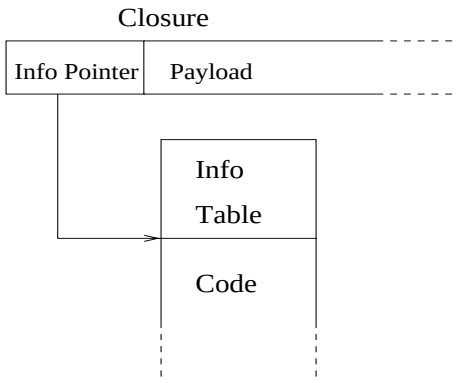



Figure 3.2: A Closure in STG-machine

### 3.2.4 TIM, the Three Instruction Machine

Another abstract machine [136] based on supercombinators has been the Three Instruction Machine or `TIM`. In `TIM` there is no spine in the heap, since `TIM` is spineless. A supercombinator expression is first flattened so that expressions do not contain nested structures. Then, the arguments of supercombinators are packaged up into tuples to which code pointers are added so that they form closures. Spineless means that closures are self-contained entities since they contain the code for functions which is applied to the arguments in the tuple. Unlike the `G-`machine in which updates are performed after each reduction step, in `TIM` updates are performed when the evaluation of a closure finishes. To support this, the stack and the closure are stored into the dump and the stack starts off empty. When the stack becomes empty again a normal form has been reached and updating happens.

### 3.2.5 STG-Machine

The main innovation of the Spineless Tagless `G-`machine or `STG` [100, 105], is that programs are not compiled into abstract machine code but into a minimal, low level, functional, intermediate language, the `STG` language. The `STG` language is so abstract that it can be given a direct operational state transition semantics to guide its translation into `C` or machine code. In the `STG` machine, heap objects are uniformly represented by closures. The first word of each closure,

called the *info pointer*, points to the *info table* containing code for executing the closure [94], see also Figure 3.2. This offers a *tagless* approach, where no tags are required to distinguish between kinds of heap objects; a jump is made to the code of the closure in the table. As with the `G`-machine, pointers to heap objects are held on the stack. Similarly to the `TIM` updates are performed, when needed, after the evaluation of a closure is finished. Similarly to the `G`-machine, the `STG` machine employs stacks that point to heap objects, the *argument stack* for closure addresses and unboxed values, the *return stack* for continuations of case expressions and the *update stack* for update frames. The `STG` machine is the engine of the `GHC` compiler.

## 3.3 Parallel Implementation of Functional Languages

The hardware expansion towards parallel computers made imperative the need for building parallel software. On the other hand, this urge complicated the issue of programming in two ways:

- each application requires the conception of a suitable parallel algorithm;
- parallel software complicates additionally the issue of reliability.

The following sections discuss the above issues within the purely functional paradigm and compare and contrast with the procedural paradigm.

### 3.3.1 Parallel Graph Reduction

Functional languages are not inherently sequential and enable parallel algorithms to be expressed in a natural way. This is due to the fact that subexpressions in a program do not have side-effects and so, it is safe to evaluate them in parallel. Again, the Church-Rosser theorem guarantees that a leftmost outermost reduction strategy can cohabit well with parallel evaluation. For example, the expression `double (square (1+3))` can be reduced in parallel as follows:

```
double (square (1+3))->
m+m where m = n*n, n = 1+3 ->
```

```
m+m where m = n*n, n = 4 ->
m+m where m = 16 -> 32
```

This demonstrates that the graph reduction model provides a simple way to express parallel evaluation. Moreover, parallel graph reduction has the following characteristics from the point of view of the semantics, discussed also by Peyton Jones [96]:

- No additional language enhancements are required to specify parallel evaluation;
- Parallel graph reduction handles the sharing of parallel tasks safely since:
  - It allows communication and synchronisation to be handled without the need of additional language constructs to ensure safety. Usually, when a thread attempts to evaluate a piece of graph, it locks it during evaluation, to prevent any other thread from duplicated the work of evaluation unnecessarily. When the particular piece of graph is evaluated, then its value can be shared at will. Thus, threads do not share any state; their states are independent of each other. They communicate only via shared constant values, which they return as results;
  - It gives deterministic results; the Church-Rosser theorem guarantees that a parallel evaluation strategy will yield the same result as any other evaluation strategy;
  - It is deadlock free;
  - Correctness proofs can be applied easily to parallel programs.

This is in sharp contrast to the traditional parallel programming model based on procedural languages because:

- They are inherently sequential and thus, they need to have additional features added to become suitable for parallel execution.
- They need synchronisation and communication techniques to prevent both deadlock and unacceptable results;

- The programmer needs to describe details of thread creation, scheduling, communication and synchronisation;
- Results obtained from parallel execution are not necessarily deterministic, since synchronisation and communication techniques are semi-safe, as has been discussed in Chapter 2;
- Correctness proofs for parallel programs become far too complicated.

From all these, one concludes that functional languages are more suitable for parallel programming than procedural ones since they offer a high grade of reliability. Furthermore, the above remarks justify also that:

> *It is only within the procedural paradigm that parallelism and concurrency need to be studied within a common framework. The purely functional paradigm separates the two issues. Parallelism has been handled within the graph reduction model in a semantically transparent way.*

### 3.3.2 Parallel Functional Programming

In a compiler of a functional language that supports parallelism, to achieve a real gain in performance, each subexpression becomes a process (thread) that runs a code segment. Multiple threads share the same heap. The run-time system does not need to provide support for synchronisation and communication between threads; as already explained this is handled by the program graph. To achieve a real gain in performance, only the appropriate subexpressions need to be turned into processes. There are two ways for turning subexpressions into processes (threads): either let the compiler infer parallelism or, more often, let the programmer annotate those subexpressions, which will give a performance gain when evaluated in parallel. An annotation is an explicit indication in an expression which changes its operational behaviour by causing reductions to take place in parallel without affecting either the result of the whole computation or its meaning. In the `ghc` compiler for example, parallelism is specified by the `par` annotation. This indicates which parts of the expression may be evaluated in parallel. A call `par x y` starts a new thread to execute `x` and then evaluates

and returns `y`. As an example, a parallel version of a merge sort algorithm in Haskell is presented:

```
msort xs =
   if length xs <= 1 then xs else
   par ys (merge zs ys) where
     ys = msort ...
     zs = msort ...
```

In each call, the main thread evaluates `zs`, the new thread evaluates `ys` and then `merge` proceeds. Similarly, other divide-and-conquer or pipelining algorithms can be easily expressed.

### 3.3.3 Distributed Machines for Parallel Functional Programming

To increase performance, parallel functional programming has employed distribution. Expressions which can be evaluated in parallel are distributed in several processors. As a consequence, a number of distributed abstract machines in functional languages have been designed to support parallelism, i.e. speeding the execution of programs and thus, increasing their efficiency. Although parallelism is beyond the scope of this thesis, design decisions made for parallel graph reduction machines influenced the design of the distributed Brisk machine, a machine model that supports distribution and computational mobility which is described in Chapter 11. Thus, four graph reduction machines are presented below, `GRIP` as a shared-memory machine, `HDG` as a distributed memory machine for a small number of processors, `JUMP` as a distributed memory machine for a large number of processors and `GUM` available for both shared-memory and distributed-memory architectures.

**HDG** `HDG` [64] is a parallel functional implementation based on a distributed-memory assumption for transputer machines implemented in Miranda[1]. It is based on the evaluation transformer model of reduction [14]. This enables a semantically sound analysis technique for specifying how functions use their

[1]†Miranda is a trade mark for Research Software

arguments and thus enables the compiler to derive parallelism. Its implementation is based on a stack which consists of a linked-list of activation records, each record points to the one which was activated immediately before it. Process switching is thus very fast; it involves saving the pointer to its current activation record. The maximum size of activation records is determined in advance [73]. The garbage collection scheme is based on Lester's algorithm [74] combining weighted reference counting and copying garbage collection techniques; the former is used for inter-process references. This has been supported by two special kinds of nodes. *Output indirections* which point to non-local objects and have a weight and, *input indirections* which are pointed at by non-local references, point to local objects and have a reference count.

Initially there is only one task, new ones are spawned by the compiler and placed in the *migratable* task pool as candidates for migration; they are removed in `LIFO` order. If the migratable task pool becomes empty then work is requested from the migratable pool of a neighbour processor. Tasks received from other processors are held in the *active* task pool; active tasks are not allowed to migrate and are removed in `FIFO` order. When it becomes empty, tasks are requested from the migratable task pool. Finally, the *blocked* task pool hold tasks which wait for a result from another task.

A five node prototype implementation compares favourably with the sequential implementation of the `LML`.

**GRIP** *Graph Reduction in Parallel* (`GRIP`) is a shared memory architecture for parallel functional programs. The hardware [101] consists of up to eighty `M68020` Processing Elements (PEs), up to twenty microprogrammable Intelligent Memory Units (IMUs) and a high-bandwidth packet-switched bus to interconnect them. This enables a two-level address space: fast, private memory is held in the `PEs` as a local address space and slower, larger memory is held in the `IMUs` as a global address space. Each of the `IMUs` holds a fixed part of the global address space and supports memory operations at a higher level than read/write instructions of conventional memory; it performs variable-sized node allocation, garbage collection and thread scheduling. Each `PE` uses its private memory as a local heap; it allocates new closures and caches copies of global closures from

IMUs.

GRIP [101, 43, 3] runs parallel Haskell programs. Its design has been adapted to the evaluation model of the STG machine [105, 100] by adding a special field in every closure to indicate the global closure of which it is a copy. If the closure has been locally allocated, this field is set to zero. A *thread* is a sequential computation to reduce a sub-graph to head normal form; initially there is one thread. *Sparking* a thread consists in placing a pointer to its closure in the *sparked thread pool* held by the IMUs. Sparking follows the *conservative* parallelism principle, i.e. a thread is sparked only when is needed. Idle PEs poll the IMUs in request for threads; IMUs set lock bits to avoid duplicating work. A blocked thread, when updated, is added to the *unblocked thread pool*. Evaluation follows the *evaluate-and-die model*[2] to avoid blocking while waiting for the child and context-switching overhead. This maximises the length of threads and prevents the system from sparking unnecessary ones. Scheduling is based on a LIFO strategy when the system load is high, whereas a FIFO strategy is employed when the load is low. Various run-time strategies for spark regulation, especially spark cut-off are employed. The results obtained show good relative speedups over the same application executing on sequential machines.

**JUMP** The JUMP [24] machine is a parallel abstract machine for the implementation of functional languages on massively parallel MIMD-computers and therefore it is a distributed-memory model for a large number of processors. It adapts some of the basic features of the Threaded Abstract Machine [29] used in dataflow languages, notably *non-blocking communication* and *hierarchical self-scheduling* into the JUMP machine [25], a variant of the STG Machine. Non-blocking communication is based on the concept of active messages [32]; an active message holds a code pointer and it is executed when it arrives at its destination, thus no buffering is required. This conforms to the spirit of the STG and JUMP machines, so a message in the JUMP machine as with any heap object is a closure containing code, the *message handler*. Since message handlers cannot be interrupted by other messages, small messages are preferred.

[2] In the evaluate-and-die model, when the parent requires the value of a closure for which it sparked a child to evaluate, the parent evaluates the closure, as if it had never sparked a child and the child will be garbage collected.

Ordinary processes are created by the SUSP instruction. *Fork objects*, i.e. exportable processes are created by the FORK instruction and stored locally in LIFO mode in the *pending process pool* (PPP); its last object, the *no-work-object* requests work from other machines. Each fork object has a *schedule method* and an *export-method*. When there is low load, the schedule method of the first object of the PPP is executed and the fork object mutates into a *notify object*. The *export method* is invoked when a processor requests work through a *work request message*. It issues then a *work message* to the requesting processor and mutates the fork object into a notify object. When a *work message* is executed in the remote processor, it creates a *remote-eval object* to evaluate the code of the fork object and send the result to its originator. *Remote-eval objects* are bound to be evaluated locally. They are placed in LIFO mode into the *ready process pool*, its first object, the *schedule-PPP-object* tail calls the first object in the PPP. Since exported objects do not export any objects they reference, references to non-local objects are split into two parts, *input objects* and *output objects*. *Output-objects* substitute remote objects in a processor and *input-objects* provide a fixed address for the referenced object in the remote processor. Similarly to the HDG machine, this scheme allows the use of Lester's [74] distributed garbage collection algorithm.

JUMP provides a scheduling hierarchy where on the lowest level the machine can execute a thread switch within around three machine instructions, therefore it tolerates efficiently any remote access latency.

**GUM** The Graph Reduction for a Unified Machine model, GUM [126] is a message-based, portable, parallel implementation for Haskell built on top of the ghc compiler (from release 0.26 onwards) as a new run-time system for it. It provides tools for monitoring and visualising the behaviour of programs. It is available for both shared-memory (Sun SPARCserver multiprocessors) and distributed-memory (networks of workstations) architectures and uses PVM as a communication harness to match processor-memory units (PEs) to available processors. GUM is message based; messages are asynchronous and may contain large amounts of graph. A GUM program initially creates a PVM manager task to control initialisation, termination and garbage collection. Initialisation consists

in spawning the required number of PEs as a number of PVM tasks which get mapped to processors. After PEs get initialised, one of them becomes the main PE and starts executing the main thread of the program. Execution in each PE is controlled by a scheduler loop in the following order: first, a garbage collection is performed, then any incoming messages are accepted, then it runs one runnable thread (if any), otherwise it requests work from another PE. Parallelism in GUM is created by par annotations; they create thunks which are added to the *spark pool*. Each PE has a collection of threads, the *runnable pool* and a collection of sparks, the *spark pool*. Threads are heap allocated objects, called Thread State objects (TSO), each pointing to a Stack Object (SO) and are not allowed to migrate. A thread locks a thunk when it evaluates it preventing multiple evaluation. Other threads attempting to evaluate the same thunk block till the evaluation finishes; then they enter the runnable pool. When there are no sparks in the spark pool, the PE launches a FISH message and sends it at random to another PE. If the PE has a useful spark it emits a SCHEDULE message to the PE that requested work, containing a spark together with some nearby graph. The originating PE unpacks the message, adds the spark in it to its spark pool and sends an ACK message which records the new location of the thunk. The original spark is overwritten with a *FetchMe* closure containing the global address of the requesting PE. A global address consists of a PE identifier together with a unique integer, the local identifier. The Global Indirection Table GIT maps local addresses to their local identifiers. Furthermore, there is a GA -> LA table which maps foreign global addresses to their local counterparts and a LA -> GA table which maps local addresses to their corresponding globals. Garbage collection is based on Lester's algorithm [74] combining weighted reference counting and copying garbage collection techniques.

### 3.3.4 Eden

Eden [13], [12] is a parallel functional language, enriched with the ability to provide explicit processes which communicate by exchanging values via unidirectional communication channels, modelled by head-strict lazy lists. As this introduces non-determinism, a predefined *merge* process abstraction is provided for *fair* merging. Dynamic reply channels to establish new connections at run-

time are provided via the *new* construct. Dynamic channel connection happens by sending the name of a dynamically created channel to a process to use it for output. Eden has been especially designed to be implemented in a distributed setting and uses the standard library `MPI` (Message Passing Interface) to map processes to available processors.

Eden is translated into the intermediate language `PEARL`, which is similar to the `STG`-language. `PEARL` is interpreted by an abstract distributed machine called `DREAM`, consisting of a set of abstract machines each of which is called *Dream* and executes an Eden process which evaluates a different output expression using a common heap that contains shared information.

# Chapter 4

# State in Functional Programming

Monads are abstract data types that act over state values together with special operations that enable them to access these state values linearly. Such linearly accessed stateful values have been called *state threads*. Monads offered a purely declarative form of update in-place and thus, a procedural style of programming within a purely functional language. State threads set the theoretical framework for handling side-effecting operations in Haskell, notably `I/O` and for providing a solution to the destructive array update problem. Additionally, state threads have been securely encapsulated [69, 102] and incorporated into compilers such as `ghc`. This framework has also been adapted to support concurrency [36] and to build reactive systems [35, 34] in Haskell.

This chapter describes this form of `I/O` as it has been incorporated into compilers such as `ghc`.

## 4.1 What is a Monad

For the purposes of a computer scientist, a monad is a triple `(M, unitM, bindM)` where `M` is a type constructor `M a` which represents a computation and `unitM`, `bindM` a pair of polymorphic functions:

```
unitM :: a -> M a
bindM :: M a -> (a -> M b) -> M b
```

where `unitM` turns a value into a computation yielding only that value without doing any other action and `bindM` applies the result of the computation `M a` to the function `a` $\rightarrow$ `M b` resulting a computation of type `M b`. By turning `M` into an abstract data type and making `unitM` and bindM the only operations available, it is guaranteed that computations happen in a linear way. This is also called *single threading.*

Additionally, `unitM` and `bindM` should satisfy the following identity and associativity laws.

```
Left unitM:  (unitM a) 'bindM' k = k a
Right unitM: m 'bindM' unitM = m
Associative: m 'bindM' (λa -> (k a) 'bindM' (λb -> h b )) =
             (m 'bindM' (λa -> k a)) 'bindM' (λb -> h b)
```

In other words `unitM` is left and right unit for `bindM` and `bindM` is associative.

The key idea behind the use of monads has been to make computations, such as `M a`, first class citizens [132, 131]. This has been achieved by using data types to express computations which involve state information, exception handling or output. If `M` is a monad, then an object of type `M a` represents a monadic action, i.e. a computation which is expected to produce a value of type `a` as well as an additional effect. Computations are distinguished from simple values through types; types indicate which part of a program may have additional effects. Although programming with monads is straightforward, it has not been realised that they could be used to mimic imperative features in a functional language, until Moggi [84, 85] observed that monads can model a variety of language features and used them to structure denotational semantics. This gave Wadler [132, 131] the insight that, using monads, functional programs

could be structured similarly, allowing impure features to be mimicked flexibly. Monads can also be combined, i.e. having a state monad with error handling, see also [61].

Ever since, monads became a second nature in the purely functional language Haskell. They offered the possibility to express in a purely functional language effects found in imperative ones such as global state, exception handling or output without spoiling the essence of algorithms [130, 131], which resulted in a procedural style of programming within a purely functional language. Additionally, since monads are abstract data types with specific operations defined on them (`unitM`, `bindM`), they have been used to indicate explicitly that a stateful value is used in a single threaded manner, which offered a new solution to the destructive array update problem. Based on the same principle, monads offered a new model for performing `I/O` in a purely functional language, which has been incorporated in Haskell [5] from version 1.3 onwards. The next section describes how the monadic approach has been adapted in compilers such as `ghc`.

## 4.2 Monadic State

For the `ghc` compiler, a generalised monadic framework has been proposed [69], which provided a new model for performing `I/O` and also supported other state-based facilities such as destructive array update. In this framework, any stateful computation or *state thread* is expressed as a state transformer of type `ST s a` which represents actions on a state type `s` returning results of type `a`:

```
ST s a :: s -> (a,s)
```

Single-threading is achieved by turning `ST` into an abstract data type. All primitive actions are made strict in the state to ensure that previous actions have completed. Actions can only be combined using the following generic operations:

```
returnST :: a -> ST s a
thenST :: ST s a -> (a -> ST s b) -> ST s b
```

where `returnST` returns a value without affecting the state and `thenST` composes sequentially two state transformers indexed by the same state type where

the second consumes the state produced by the first. Operation `returnST` corresponds to `unitM` and `thenST` corresponds to `bindM` in the definition of monad. Furthermore, the use of constructor [60] classes[1] provided monadic classes as a set of `return`, `>>=`, `>>` operations common to a number of related types. These types are all container types, that is they contain a value or values of another type and any instance of the above class should satisfy the above mentioned monad laws.

```
class  Monad m  where
    (>>=)   :: m a -> (a -> m b) -> m b
    (>>)    :: m a -> m b -> m b
    return  :: a -> m a
```

### 4.2.1 External State Types (`I/O`)

All external state-based facilities, i.e. `I/O` and foreign procedure calls, are captured into a single state type called `RealWorld` that represents the `I/O` state of programs and all access to these facilities occurs in a single global state thread, so that a linear ordering is put on all procedure calls. A simple approach to `I/O` could use the type:

```
type IO a = ST RealWorld a
```

Additional primitives `getChar` and `putChar` have also been provided to read and write characters from the standard input and output respectively, which work only with the `I/O` type.

```
getChar :: IO Char
putChar :: Char -> IO ()
```

Primitive `getChar` is an action which reads a character from the standard input and returns this character, whereas `putChar c` prints character `c` to the standard output and returns no value.

The monadic `I/O` approach allows for both automatic and manual error handling.

[1] Constructor classes are a natural generalisation of type classes [42, 133] in Haskell, which combined overloading and higher-order polymorphism.

```
type IO a = ST RealWorld (Either IOError a)
data Either a b = Left a | Right b
catch :: IO a -> (IOError -> IO a) -> IO a
try :: IO a -> IO (Either IOError a)
```

Within the monadic framework, `I/O` actions are composable. Bigger programs can be formed by composing simpler ones using the monadic combinators. As an example, the definition of the Haskell function `getLine` which gets a line from the standard input is given.

```
getLine :: IO String
getLine = getChar >>= \c ->
  if (c == EOF) || (c == '\n' ) then return ""
  else getLine >>= \line -> return (c : line)
```

The value of the entire program is a single action, called `main`:

```
main :: IO ()
main = putChar 'a'
```

To execute a program, `main` is applied to the `RealWorld` state representing the current state of the world, the result `RealWorld` is extracted and any changes the program assumes are applied to it in-place. In-place update is feasible since `IO` is an abstract data type with operations `return` and `>>=` to act upon. This ensures that procedure calls are single-threaded, each one consuming the `RealWorld` state produced by the previous one. Thus, `I/O` operations can be applied immediately to the real world and thus, no state argument needs to be passed explicitly, the whole state of the machine may be used instead and all updates may be performed in-place.

To complement the picture, here is a more complex `I/O` example:

```
main :: IO ()
main = do
putStrLn "Enter a file name"
putStr "fileName: "
fileName <- getLine
```

```
putStrLn "Enter a word to be searched"
putStr "Word: "
word <- getLine
contents <- catch (readFile fileName) report
putStrLn ("Result: " ++ find word contents)
main

report :: IOError -> IO String
report err = do
return "File not found"

find :: String -> String -> String
find = ...
```

### 4.2.2 Internal State Types

All internal facilities such as incremental arrays are bundled into a polymorphic state type `s` which notionally represents the program's heap. All access to internal state types happens via references. These are provided as pointers to ordinary values or to special state types such as arrays. A reference `MutVar s a` denotes an updatable location in a state of type `s`, which contains values of type `a`. The following primitive operations are provided that create, extract the value and change the value of such state locations:

```
newVar :: a -> ST s (MutVar s a)
readVar :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

The state type `s`, as well as representing the heap, is also used for encapsulation to ensure safety of state thread accesses. For internal facilities, the primitive `runST` has been provided, which has a rank-2 polymorphic type, i.e. a polymorphic function is allowed to be used as a polymorphic type:

```
runST :: (∀ s. ST s a) -> a
```

Primitive `runST` takes a complete action sequence as its argument, creates an initially 'empty' state, obeys the actions and returns the result discarding the

final state. To ensure safe encapsulation of state threads, function `runST` is not given a usual Hindley-Milner type, i.e. universally quantified over both `s` and `a`. Instead, it is given a rank-2 polymorphic type [102], universally quantified only over the type of `a`, leaving the state type value `s` polymorphic. Thus, each internal state thread has a separate polymorphic state variable which on the one hand it represents the state and on the other hand it 'tags' the references in it with this variable. As a result, for each state thread, the enclosing `runST` limits its scope and prevents its references from escaping, or references belonging to other state threads from being used within it. By forcing the state argument to be polymorphic, `runST` can only be used for internal facilities, since they refer to a polymorphic state type, and not for `I/O` which is applied to the specific state type `RealWorld`. This enforces a single `I/O` state thread, taken to exist when the program begins, by preventing further unrelated `I/O` state threads from being created, which could destroy referential transparency.

The following "illegal" example illustrates that rank-2 polymorphic types enable the secure encapsulation of state threads:

```
bad1 = writeVar (runST (newVar 41 `thenST` \p -> returnST p)) 42
bad2 = newVar 41 `thenST` \p -> runST (writeVar p 42 `thenST` ...)
```

In the first example, the expression `newVar 41 `thenST` \p → returnST p` has type `ST s (MutVar s Int)`. This can be generalised to `∀s . ST s (MutVar s Int)`. However, `runST` cannot be applied to this type, because the types don't match, as the type of `runST` would require a type signature as follows:

```
runST :: (∀ s' . ST s' (MutVar s Bool)) → MutVar s Bool
```

Similarly, the second example is "illegal" under 2nd order polymorphism types.

**Arrays** Incremental arrays have been implemented as arrays of references, as an array can be thought of as a collection of updatable locations in a state of type `s` which contain values of type `elt`. The state variable here is also polymorphic, representing the program's heap and any access to the array happens via them. The following operations to create an array, extract the value in the ith position, change a value in the ith position are provided:

```
newArr :: Ix x => (i, i) -> elt -> ST s (MutArr s i elt)
readArr :: Ix i => MutArr s i elt -> i -> St s elt
writeArr :: Ix i => MutArr s i elt -> i -> elt -> ST s ()
freezeArr :: Ix i => MutArr s i elt -> ST s (Array i elt)
```

Function `newArr` allocates a new array to the state by allocating a new array reference, `readArr` takes an array reference uses the state to map the array reference to its value, `writeArr` takes a reference and a value and transforms the array so that it map the reference to the given value. Function `freezeArray` turns a reference to an array into a Haskell array. It takes a reference, looks up the state and returns a copy of what it finds along with the unaltered state.

## 4.3 Strict and Lazy State

Both strict and lazy versions of `thenST` have been proposed:

```
thenST :: ST s a -> (a -> ST s b) -> ST s b
thenST m k s = case m s of
   (x, s') = k x s'


lazyThenST :: ST s a -> (a -> ST s b) -> ST s b
lazyThenST m k s = k x s'
   where (x, s') = m s
```

Launchbury [70] experimented using `lazyThenST` with internal state types. He argues that *lazy state threads*, i.e. stateful computations that use `lazyThenST` match better the lazy semantics as they preserve laziness in the presence of imperative actions. Additionally, lazy state threads are more adequate with internal state types, because the state is discarded at the end of the computation.

The strict version of `thenST` has been suggested for use with `I/O`. Although it has been claimed [69] that the reason of using the strict version of `thenST` with `I/O` is efficiency, research in this thesis using the lazy version of `thenST` with `I/O`, see also Chapter 6, offers a new insight into this subject:

> *The strict version of* `thenST` *apart from efficiency, plays a more important role. It is the strict version of* `thenST` *which allows infinite*

`I/O` *sequences to produce results.*

For example, an infinite printing loop can produce output only when the strict version of `I/O` is used. As a counterexample, consider an infinite printing loop using the lazy version of `thenST`:

```
loop = putStr 'a' >> loop
```

Tracing the operations of this program gives:

```
loop s0 = loop s1 where ((), s1) = putChar 'a' s0
loop s1 = loop s2 where ((), s2) = putChar 'a' s1
...
```

It is clear that the `I/O` state becomes in each step a larger unevaluated expression and nothing forces any `I/O` operations to happen. By using the strict version of `thenST` with `I/O` in an infinite `I/O` sequence, it is ensured that the state gets evaluated at each step and any side-effecting operations are produced. This proves that the strictness of `thenST` operator plays a more vital role than increasing efficiency. The use of the strict version of `thenST` is not vital in the case of a finite sequence of `I/O` operations; when the final `I/O` state is produced, it is updated in place, which causes all the side-effecting operations to happen.

### 4.3.1 Interleaved and Parallel Operations

In the previous paragraph it has been explained why the strict version of `thenST` has been used with `I/O`. On the other hand, this creates further problem.

> *Using the strict version of* `thenST` *[106, 69] is problematic in the case of interleaved* `I/O` *operations such as reading lazily the contents of a file to be expressed in Haskell. This forces a file to be read as a whole, preventing other actions to be performed until the reading finishes.*

To enable the contents of the file to be read lazily, a special monadic primitive `interleaveST` [69], has been discussed for `ghc`. This primitive splits the state into two distinct parts, one corresponding to the state of the file and the other

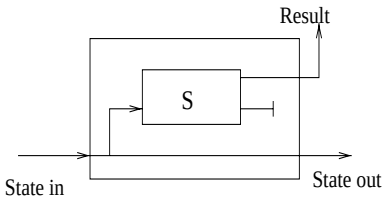



Figure 4.1: The interleaveST Primitive

to the rest of the `RealWorld` state, so that operations onto the file can be performed concurrently with operations to the rest of the `RealWorld` state, see also Figure 4.1. The disadvantage of such a primitive is that the state of the file and the rest of the state are not independent and so actions on the file and actions on the main state may produce non-deterministic effects.

Since `interleaveST` is an unsafe primitive, in all Haskell systems the following primitive has been provided implemented in `C`.

```
hGetContents :: Handle -> IO String
```

This takes the identifier of a file and returns its contents as a string. The contents are read lazily in demand for the result.

One of the main contributions of this thesis is that it offers a solution to the lazy file reading problem which is safe. This has been achieved using the

As it will be demonstrated in chapter 6 of this thesis, the use of the lazy version `lazyThenST` offers a solution to the lazy file reading problem.

## 4.4 Monadic I/O vs Dialogues

Certainly the monadic `I/O` offers important advantages over the stream `I/O` model. As it is mentioned by Wadler and Peyton Jones [106] it is:

- composable;
- efficient;
- extensible.

On the other hand, the observation that the strict version of `thenST` is required in the monadic `I/O` for coping with infinite `I/O` sequences, reveals a semantic weakness. Take again the `loop` program in both stream and monadic `I/O` respectively:

```
type Dialogue = [Response] -> [Request]
loop :: Dialogue
loop resps = Getc :
             if (a == eof)
             then []
             else Putc a :
                  echo (loop 2 resps)
             where
               OhCh a = resps!!1


loop :: IO a
loop = putChar 'a' >> loop
```

Stream `I/O` provides infinite `I/O` sequences as it can be simulated directly whereas monadic `I/O` cannot. In the case of the Stream `I/O` model, any infinite `I/O` sequence is applied to a bottom ($\perp$) list of responses, the first request is extracted and the first response calculated, then the bottom ($\perp$) list of responses is updated in place to form a new list of responses. The `I/O` operations are then applied to this updated list of responses, the second request is extracted etc.

```
req0  = main ⊥                  (req0 = Getc)
resp0 = head resps               (resp0 = Success a)
req1  = main (resp0 : ⊥)         (req1 = Putc a)
resp1 = (resps !! 1)             (resp1 = OkCh a)
resp2 = main (resp1 : resp0 : ⊥)
...
```

Unlike stream `I/O`, the monadic `I/O` cannot be simulated directly because the `I/O` state becomes in each step a larger unevaluated expression and nothing forces any `I/O` operations to happen.

```
loop w0 =  loop (snd (putChar 'x' w0)))
           loop (snd (putChar 'x' (snd (putChar 'x' w0))))
           ...
```

In other words, with monadic `I/O` it is not possible to simulate one by one the operations involved in an infinite `I/O` program, which semantically means that loop $\perp = \perp$. One can only trace the individual operations as above. Of course the use of `lazyThenST` is fine in the case of finite `I/O` sequences, where once the final state is obtained, the previous ones are also obtained backwards and the `I/O` operations applied to the real world.

This semantic weakness of the monadic `I/O` becomes more serious in the Brisk monadic framework described in Chapter 6 where the use of `lazyThenST` tends to be vital for supporting deterministic concurrency.

# Part II

# Functional Concurrency

# Chapter 5

# Deterministic Concurrency

The contribution of this chapter is twofold. First, it introduces *deterministic concurrency* [23, 51], a purely declarative approach to concurrency which does not allow non-determinism to appear at a user level. This extends the demand driven model, used as the operational basis of functional languages, into one with multiple independent demands in the form of threads that do not communicate between them. To avoid the non-deterministic effects caused by the use of a non-deterministic merge operator, necessary during thread communication, threads communicate only via values they produce as results. This model of concurrency has been initially introduced in Haskell 1.2 on top of the stream `I/O` model and proven expressive enough to provide a complete single-user multi-tasking working environment including shells, file managers and other operating system features.

This chapter attempts to incorporate deterministic concurrency in Haskell 1.4 on top of the monadic `I/O` framework [106]. Deterministic concurrency takes the form of truly independent threads that act on disjoint parts of the `I/O` state, called *substates*. *State splitting* is also introduced, a mechanism which assigns at creation of each new thread a substate, disjoint from other substates, to act upon. This approach enables a concurrent setting where several independent threads access the outside world, i.e. the `I/O` state of the program. Under such a deterministic approach to concurrency, a processor can run one or more `I/O` performing operations without exhibiting any user-visible non-determinism.

## 5.1 Introduction

Functional languages have been criticised in the past as inadequate to provide concurrency. This is because concurrency introduces non-determinism as a consequence of timing dependencies during process creation and communication, which is incompatible with their philosophy. A number of attempts to introduce concurrency into purely functional languages has been undertaken. In this chapter, a survey of these attempts is presented.

Additionally, this chapter introduces *deterministic concurrency*, a purely declarative approach to concurrency which fits the purely functional paradigm. This allows `I/O` performing processes to be handled in a semantically transparent way by extending the demand driven model used as the operational model of functional languages into one with multiple independent demands. This chapter explains how deterministic concurrency has been introduced in Haskell 1.2 on top of the stream `I/O` model. Moreover, it is discussed how it has been adapted to be suitable for the monadic `I/O` framework and how it can be made more expressive by introducing a deterministic form of communication.

### 5.1.1 Referential Transparency

In all programming languages, programs are given meaning via both the *denotational* and the *operational* semantics. The denotational semantics study the values of programs whereas the operational semantics study their behaviour. Purely functional languages are the only programming languages where the values of programs correspond to their behaviours. In other words, purely functional languages exhibit the principle of *referential transparency*.

> *Referential transparency is the principle of purely functional languages that ensures that the denotational semantics match exactly the operational semantics and thus, the mathematical properties of programs are also properties about their behaviour [50].*

This implies that expressions have no other effect than their values themselves. As a consequence, any function gives always the same results when called with the same arguments. This enables equational reasoning, i.e. the use of equations to replace subexpressions with equivalent ones and offers confidence

in the correctness of programs. As a result, purely functional languages offer a particularly clear and expressive approach to programming and their clean semantics allow for proof of program properties. This makes programs easy to adapt, debug and maintain. At the implementation level, this feature of functional languages emerges as a lack of uncontrolled update. Memory locations are normally only updated in the sense that unevaluated expressions are replaced by their values, and thus they can be freely shared. Updates in which values are changed are regarded as optimisations in which memory is reused when sharing is known to be absent. Issues of consistency in accessing updatable shared values which beset procedural systems do not arise.

## 5.2 Concurrency in Functional Languages

To provide concurrency, programming languages have had concurrency primitives added into them which create, control and allow processes to cooperate. However, as it has been explained in Chapter 2, concurrency primitives introduce non-determinism during thread creation and communication. Multiple threads may share the same state with non-deterministic interleaving of accesses. Furthermore, threads communicate by passing messages and non-deterministically merge them to a common destination.[1] As a consequence, concurrent systems exhibit non-deterministic behaviour. This can manifest itself as undesired behaviour under certain, usually rare, sequences of events. It also adds an extra burden of complexity onto the programmer, making it very difficult to assert such systems as reliable and resilient.

Moreover, any attempt to add non-deterministic features directly into functional languages is potentially damaging. Non-determinism has adverse effects on referential transparency and so, it destroys the clean mathematical properties of functional languages since equational reasoning cannot be applied [27]. This is the reason why functional languages had been in the past heavily criticised as inadequate for building concurrent systems such as interactive programs or programs which need to communicate with external resources.

[1] In stream based programming the term *merge* is used instead of *choice* or *select*.

### 5.2.1 Non-determinism in Purely Functional Languages

Similarly to procedural programming languages, early attempts to introduce concurrency into purely functional languages considered a set of processes statically defined by the program. This in turn, immediately introduced the problem of merging streams of information and implied the need for a non-deterministic merge operator. On the other hand, this is additionally problematic in a purely functional language as it is impossible to define a `merge` facility:

```
merge "abcd..." "ABCD..."
"aBbAcCDd.."
```

Such a facility cannot be expressed as a function; it returns different results when called with the same arguments, since the order at which items appear in the input streams can be totally arbitrary as it depends on accidental timings.

In order to focus on the semantic problems that emerge from the need of a non-deterministic merge, McCarthy [81] introduced a binary choice operator `amb` as a function which returns either of its arguments. The expression `a ‘amb‘ b`, returns either `a` or `b` according to which is evaluated first. Such an angelic operator destroys equational reasoning since the expressions:

```
(x + x) where x = (4 ‘amb‘ 5)
(4 ‘amb‘ 5) = (4 ‘amb‘ 5)
```

are not any more equivalent.

To bridge the gap between non-determinism and the mathematical foundations of functional languages, Burton [16] proposed a solution that uses pseudo-data. Each program is supplied along with its input with an extra argument, an infinite, lazy, binary tree of values. Every time a non-deterministic choice is made, it is recorded. When the same function with the same arguments needs to be evaluated again, the choice is recalled and instead of re-evaluating the expression, the recorded value is used. This idea, although it preserves referential transparency, destroys the Church-Rosser property[2] because the results depend now on the order of evaluation. Furthermore, this idea turns out to be

[2]According to the Church-Rosser property (Theorems I and II [120, 98]), any reduction sequence should lead to the same normal form.

rather impractical, e.g. in an interactive application it is difficult to record all the possible choices.

Søndergaard and Sestoft [117, 116] studied the impact of non-determinism in functional languages independently from concurrency by adding directly non-deterministic operators into them. This is not explored further, as the aim of this thesis is to exclude non-determinism in a concurrent setting.

### 5.2.2 Purely Functional Operating Systems

In the 80s, a large amount of research has been carried out in building purely functional operating systems. Functional languages have always suffered as having a guest status within procedural operating systems. Procedural operating systems exhibit non-deterministic behaviour either through aliasing where a file can be accessed via more than one different filenames, or through the use of a select system call.

First Henderson [47] studied a range of systems which encapsulate basic operating system design aspects. Initially, he considered the simplest possible aspect of an operating system as a program that accepts input from a keyboard and displays output to a screen and also a simple database for saving and recalling files. Several extensions to this simple setting have been discussed, from a system that allows two users with separate keyboards to share the database to a more sophisticated system with multiple filestores and editing facilities. The common characteristic of all these extensions is the need for a non-deterministic merge operator called `interleave` to interleave input from different users. Henderson admitted that such an `interleave` operator destroys the nature of functional languages and recommended restricted use of it. On the other hand, he admitted that a non-deterministic primitive is indispensable for modelling non-deterministic behaviour.

Jones and Sinclair [62] followed Henderson's work as a basis [47] and presented the design of a single-user multiprogramming operating system based on streams. Such a design aimed at demonstrating that streams can be exploited to fulfil the role of interactive communication channels between programs and external devices. Again a non-deterministic merge operator was necessary. Rather

than an `interleave` operator, a `merge` one has been used: [3]

```
merge :: [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x!xs) ys = x ! merge xs ys
merge xs (y!ys) = y ! merge xs ys
```

This `merge` operator is also non-deterministic because timing dependencies define which `merge` alternative will be executed next but it gives implicit time determinism since the choice is time dependent rather than random as with the `interleave` one discussed by Henderson [47].

Another alternative approach was presented by Stoye [121]. He considered a collection of processes forming an operating system each with a single input and a single output stream. To avoid a non-deterministic merge, processes do not send messages to each other. Instead, processes communicate by sending messages via a central service, the *sorting office*. Non-determinism is pushed to the sorting office, leaving processes referentially transparent, so equational reasoning can always be applied to each one separately. This makes the system non-deterministic as a whole. This idea has been considered the best available compromise and has been used also by Turner [128] for the `KAOS` project.

### 5.2.3 Timestamping

Another solution to the problem of merging streams of information would be to turn a non-deterministic merge into a deterministic one by attaching timing information to items coming from the input streams. This approach has been used in the programming language `RUTH` [46]. In `RUTH`, communication between processes is modelled by streams of timestamped values. Timestamping is achieved by giving as input to programs a tree of integers which is lazily evaluated as the program executes to produce timestamped node values. A merging mechanism has been defined:

[3] The notation `!` denotes head strict cons. The expression $e_1$ ! $e_2$ forces the evaluation of $e_1$ but suspends the evaluation of $e_2$, building a stream whose first message is a value of $e_1$ and whose subsequent messages are described by $e_2$.

```
Ready : chan x num -> boolean


merge : chan x chan x num -> chan
merge lambda(ch1, ch2, d, n).
 If Ready(ch1,n)
 Then If Ready(ch2,n)
     Then If Time(ch1)≤Time(ch2) Then ans1 Else ans2
     Else ans1
 Else If Ready(ch2,n) Then ans2 Else(ch1,ch2,d,n+k)
 where
 ans1 = ConsCh(HeadCh(ch1),at(Time(ch1)+d),
        merge(TailCh(ch1),ch2,d,n+k))
 ans2 = ConsCh(HeadCh(ch2),at(Time(ch2)+d),
        merge(ch1,TailCh(ch2),d,n+k))
 k = a constant number
```

It is obvious that timestamp comparison in RUTH gives also non-deterministic results. If messages are available in both channels then the message with the earliest timestamp is forwarded, otherwise the first available message will be forwarded even if the delayed message might have an earlier timestamp, which fosters non-deterministic results.

This thesis presents a timestamping mechanism that is not based on accidental timings of arrival of messages in the input channels, but takes into account only the time of creation of the messages. A message is not accepted until it has been determined that no earlier messages can arrive in any of the input channels. This is presented in Chapter 7.

### 5.2.4 Concurrent ML

Reppy's Concurrent ML (CML) [110, 111, 86] is a concurrent version of SML [71]. CML extends SML with facilities for providing multiple thread creation, dynamic channel creation and first-class synchronous operations. This work is different from the ML-threads package [28] which provides threads, locks and condition variables, aiming at building low-level O/S system devices which support parallelism rather than concurrency.

The most important contribution of CML is the provision of first class synchronous primitives in a higher order language. Reppy [110] argues that synchronous primitives don't cohabit well with abstraction. Wrapping a client-side protocol into a function offers an attractive abstraction. On the other hand, this abstraction does not fit well with selective communication. When a client blocks on a call, selective communication is required to respond to other communications otherwise the whole program will block, but this is not possible because the function abstraction has hidden away the synchronous aspect of the protocol. To overcome this mismatch, Reppy observed that selective communication can be decomposed into four stages: the individual I/O operations, the actions associated for each operation, the non-deterministic choice and the synchronisation. An abstract data type event has been provided together with operations for forcing synchronisation on this type corresponding to the above mentioned phases. Thus, receive and transmit operations build event values that represent channel I/O operations, wrap binds an action to an event value, choose composes event values into non-deterministic choice and sync forces synchronisation on this data structure corresponding to the above mentioned phases. This allows to provide first class synchronous operations without breaking the function abstraction. On the other hand, as Peyton Jones [36] argues CML code should be modified in each program according to the expected result.

Overall, despite the important insights in building selective communication into functional languages, CML also adopts the non-determinism that derives out of selective communication and builds a higher order imperative facsimile of concurrency.

### 5.2.5 Facile

Facile [40], [124] is another extension of ML that provides both communication and multiple threads of control. The basic concept in Facile is a *behaviour* which is similar to an event in CML. Behaviours are used both to support communication without losing abstraction and to provide multiple threads of control. From the point of view of determinism, Facile shares the same drawbacks as CML.

### 5.2.6 Concurrent Haskell

Unlike ML, Haskell is a purely functional language without any side-effects and therefore, introducing non-determinism has been harder to accept. Nevertheless, despite the semantic incompatibilities between purely functional languages and non-determinism, Haskell [5] has been similarly adapted to support concurrency by having concurrency primitives added [36]. Concurrency has been built on top of the monadic I/O framework by providing a mechanism for creating dynamically new processes.[4] New *threads* (lightweight processes) can be created by the monadic primitive:

```
forkIO :: IO () -> IO ()
forkIO m s = s' `par` return r s
   where (r,s') = m s
```

In a call `forkIO m`, a new thread is created to evaluate action `m` independently from the main thread. The fact that parent and child threads mutate the same state authorises non-deterministic effects, i.e. two processes can access simultaneously the same file and thus, traditional synchronisation mechanisms have been employed to keep non-determinism to a minimum.

Synchronisation has been facilitated through a primitive type `MVar` which can be thought of as a synchronised version of an updatable location `MutVar` introduced in Section 4.2. A primitive type `MVar a` is similar to a lock variable and represents a mutable location which is either empty or contains a value of type `t`. The following primitives have been provided to create, access and alter `MVars`:

```
newMVar :: IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar MVar a -> a -> IO ()
```

Primitive `newMVar` creates a new `MVar`, `takeMVar` blocks until the location is non-empty, then reads and returns the value of an `MVar`, `putMVar` writes a

[4] The term *state thread* has been used in Chapter 4 to denote stateful computations that are linearly accessed. *Threads* (lightweight processes) in Concurrent Haskell have been built on top of state threads, as they are stateful computations themselves. For the rest of this chapter the term thread is used to denote lightweight processes.

value into a specified location. These primitive operations on mutable variables enabled synchronisation and communication to be handled as conveniently as I/O.

Based on mutable variables (MVars), Concurrent Haskell provided facilities for mutual exclusion. Also MVars have been used as basic building blocks upon which further concurrency mechanisms can be built. For example, mutable variables MVars have been extended into channel variables (CVars) which handle acknowledgements from the consumer to the producer. Such channel variables offered a rendezvous solution to the producer-consumer problem. Channel variables have also been used to provide an alternative solution to the producer-consumer problem using an unbounded buffer. A more detailed description of all these abstractions can be found in [36].

Unlike CML, no first class synchronous primitives have been provided in Concurrent Haskell. The functionality of a select primitive has been achieved by a mechanism which uses a shared structure protected by synchronisation primitives such as a mutable (MVar) or a channel (CVar) variable. This approach enables one to built select-like mechanisms equivalent to both an alternative and a repetitive command in CSP. This is successful provided that the separate threads have independent I/O, so that if one thread blocks in an attempt to read from a channel, the other threads may continue. A similar approach has been used in the programming languages PICT [107], based on the $\pi$-calculus.

Still, the main shortcoming of Concurrent Haskell is that it introduced non-determinism. To restore semantic properties, non-determinism is restricted to the interaction with the outside world e.g. I/O. This offers a compromise called *referentially transparent non-determinism* [34]. In other words, non-determinism is pushed out of the functional program into the I/O actions and I/O state leaving the functional program referentially transparent. For example, a mergeIO operator has been provided which merges non-deterministically streams of information:

```
mergeIO :: [a] -> [a] -> IO [a]
```

Primitive mergeIO takes two lists of events and merges them according to whichever item appears first on either of the two lists. Its monadic nature

guarantees that it is referentially transparent, since its non-determinism is associated with the `I/O` state, leaving the functional program referentially transparent. This means, that two different calls of `mergeIO` with the same arguments are assumed to be applied to different implicit states, so the same result would not be expected. Thus, `mergeIO` retains the internal referential transparency of the language, while it exhibits non-deterministic behaviour. Operators such as `mergeIO` have been used in implementing Haggis [35], a graphical user interface for the functional programming language Haskell [5] built on top of the `ghc` compiler. Although this approach to non-determinism restored the semantic properties of functional programs, programs still suffer from the damaging effects of non-determinism; their externally visible behaviour remains non-deterministic and thus less predictable, repeatable and composable.

## 5.3 Deterministic Concurrency

In Chapter 3, it has been described that purely functional languages offered a non-determinism free model for parallelism. This has been achieved by providing threads which do not share any state; their states are independent of each other. Thus, in a purely functional language parallelism does not need to have a guest status within the general concurrency framework and can be studied as a concept on its own right. Could the same be true for concurrency? Could concurrency benefit from this model?

The need for a truly declarative approach to concurrency, prompted further research in the functional programming community. Holyer and Carter [23, 51] decided to study concurrency under a totally different perspective by taking into account the advantages that the purely functional paradigm offered to parallelism. To achieve this, instead of trying to fit functional languages within the philosophy of concurrency they went back to re-examine the issue of concurrency itself. They observed that concurrency is highly influenced from being studied within the context of procedural languages because their philosophy leads to a *data driven* approach to it. This is in contrast to the non-strict functional programming evaluation model which is *demand driven*. In this model, expressions are evaluated on an "as-needed" basis and computation evolves in

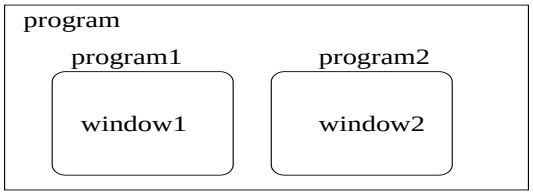



Figure 5.1: Inherent Concurrency

order to satisfy a single requirement or demand, that of providing some output to the external world. They showed that a deterministic approach to concurrency can be achieved by switching from a conventional *data driven* approach to a *demand driven* approach. This demand driven approach offers a purely declarative model of concurrency which is called *deterministic concurrency*.

### 5.3.1 Data Driven Design

A more careful examination of the conventional concurrency model reveals that it is *data driven*. Tasks a program carries out are driven by independent input streams to it and the program must be able to cope with events coming from these multiple input streams, responding to each whichever event arrives first. This requires to non-deterministically merge the events from multiple input streams into a single stream. As already discussed in Section 2.5.1, the merging may be external to the program, in the sense that the program accesses a single queue of events from different sources. Alternatively, the merging may be internal in the form of a procedure call, allowing a program to accept whichever event arrives first on a given set of input streams. In both cases, a non-deterministic merge has fatal consequences on both functional programming languages and the behaviour of programs. In Figure 5.1 for example, key presses or mouse clicks from either window are regarded as forming separate input streams. Suppose the program is idle, and that you press a key in one of the windows to communicate with the program. The program needs a mechanism to be able to respond to an item appearing on either of the two input streams, whichever appears first, which requires a non-deterministic merge. To our knowledge, all existing operating systems and graphical user interface designs are based on a data driven approach.

### 5.3.2 Demand Driven Design

To deal with inherently concurrent behaviour without non-determinism, Holyer and Carter [51, 23] were based on the lazy, demand driven model of evaluation of functional languages. They suggested that a purely functional approach to concurrency can be obtained by switching from a conventional *data driven* approach to concurrency into a *demand driven* one. Unlike the data driven approach, where attention focuses on multiple input streams, under the demand driven approach attention focuses on multiple output streams. Separate output streams are taken to correspond to multiple independent demands from the outside world which drive the program via separate threads (lightweight processes). Each thread decides deterministically which input stream to read from; this does not require a non-deterministic merge. In other words,

> *a demand driven approach to concurrency consists in extending the usual single demand model of execution of functional languages into one with multiple independent demands.*

This is less expressive than conventional concurrency, but that is exactly the point; non-deterministic behaviour is outlawed. Any timing races for the contents of the shared graph do not introduce any observable non-determinism. Nevertheless, it is still possible for a program to respond to each event, as it arrives. If, for example, each input stream has a thread which is suspended because it is waiting for input from that stream, then the first event to arrive will simply wake up the relevant thread and allow it to continue processing.

As an example, take the program of Figure 5.1. Under the demand driven approach, the two windows correspond to two independent demands on the program, one from each window asking for more output to be produced. The program can be thought of as having two threads running. When the program is idle, both threads are suspended waiting for input from their respective windows. A key press or a mouse click in one of the windows wakes up the relevant thread and causes some output for this window to be produced. In contrast to the data driven approach, where the program needs to merge the input streams, the demand driven approach can be implemented without merging the two streams.

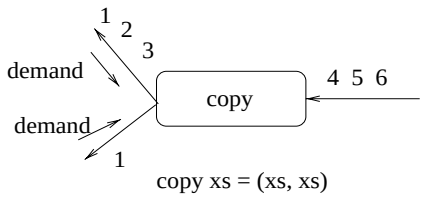



Figure 5.2: The Semantics of Multiple Independent Demands

**Semantics** The semantics of multiple independent demands can be explained by the function `copy`, as can be seen in Figure 5.2, which illustrates that it is feasible in a purely functional language to have two or more output streams which do not interfere with each other and produce results at a different rate. The `copy` function can be defined as follows:

```
copy inp = [inp, inp]
                          ↓          ↓
copy (1:2:3:inp') = [(1:2:3:inp'), (1:2:3:inp')]
```

In this example, demands from the two outputs are responded to independently from each other, so that each output is a normal function of the input. The marker ↓ indicates that three items have already been produced with respect to the first demand, while only one has been produced with respect to the other, since items at the second output stream are produced at a slower rate, see also Figure 5.2. When the next input item will be read in, it will immediately appear with respect to the first demand, but it will have to be buffered with respect to the other.

### 5.3.3 Deterministic Concurrency Model

Under the demand driven approach to concurrency, multiple external requirements are served in such a way that they don't conflict and interaction of inputs and outputs happens in the right order i.e. the timing of the arrival of input items affects only the timing of production of output items, but cannot affect the relative timings or the values produced. Referential transparency is preserved to the extent that output values and their ordering become fully defined by the code and the input values and their ordering. Programs perform `I/O` without

any externally visible non-determinism. The result is a purely declarative form of concurrency called *deterministic concurrency* [23] in which:

- combining processes into single concurrent programs does not introduce any dependencies between them which did not previously exist;
- externally visible concurrency is associated with independent outputs from a program;
- as well as preserving referential transparency inside a program, the externally discernible `I/O` effects are also deterministic and can be reasoned about;
- programs can be used as components and combined into larger programs which are still referentially transparent;
- operating system and network facilities can also be made deterministic and brought inside the model.

Deterministic concurrency can thus be seen as extending the expressiveness of functional languages so that they deliver the extra expressive power of concurrency of the operating system primitives, without their non-determinism. Although such a demand driven approach to concurrency is more restricted than the traditional form of concurrency, it enables the behaviour of reactive systems to be expressed in purely functional terms.

In this model, the initial thread of execution creates threads to satisfy demands to the program. When there is no more demand on the main thread it terminates. The program though does not terminate, since other threads may still be running; it terminates when all threads terminate.

Fairness of threads is ensured by time-slicing. Thread switching occurs when a thread's time-slice is over. The idea of multiple independent demands produces systems where deadlocks caused by accidental timings of events can be completely avoided. If a thread cannot produce any more output because its value became bottom, other threads are not prevented from continuing; when the time-slice of the blocked thread finishes another thread takes over so that eventually all threads progress and liveness is guaranteed. Additionally, collision when a thread engages a long computation without requiring input is prevented.

It is obvious that in such a demand driven model, there is no need for a non-deterministic merge. Any program can have separate action sequences, each driven by its own demand and each capable of cooperating with external services. This results in concurrent `I/O`. If one thread is suspended waiting for input, the other threads should be able to continue. This implies that each thread should have its own independent `I/O`, so that if one thread blocks in an attempt to read from a channel, the other threads may continue.

However, it is still possible for a program to respond to whichever event arrives first on any of the input streams. This is important when all concurrent threads are currently blocked waiting for input. The first item to arrive causes the relevant thread to be woken up and to produce some more output. This implies the need for a `select` facility in the implementation, for responding to whichever item arrives first, but its non-deterministic power is not available at the user-level. In practice, one can think of a central `I/O` controller which keeps track of all input channels. If a number of threads are waiting for input on separate input channels, the controller responds to the first input to arrive regardless of which item it arrives on. This amounts to a non-deterministic merge. However, all the controller does is to wake up the relevant thread to produce some output. Thus, the program as a whole is not able to produce different results depending on the relative timings at which items arrive on input channels. In such a setting, the timing of the inputs affects only the timings of the outputs, as with sequential programs. Under such a deterministic concurrency approach, a processor can run one or more `I/O` performing operations without including any user-visible non-determinism. Concurrent `I/O` under the monadic approach is also discussed in Chapter 6 in Paragraph 6.4.3.

Deterministic concurrency [23, 51] has been first introduced in Haskell upon the `Dialogue` [90] model. The `Dialogue` model is a stream based `I/O` model used as standard in early versions of Haskell. To investigate the expressiveness of deterministic concurrency, Holyer and Carter [23, 51] considered simple systems with a fixed collection of processes, with no shared state and without any non-deterministic feature available to the programmer. Processes communicated only via a statically defined network of streams. Experiments showed that

*deterministic concurrency is expressive enough to implement a complete single-user multi-tasking working environment including shells, file managers and other single-user operating system features. This proved that most of the non-determinism apparent in traditional concurrent systems is unnecessary; it results from the use of non-deterministic primitives in programming applications which are not inherently non-deterministic. This also proved that non-determinism which relies on the programmer is not good enough and can be eliminated with minor redesigns instead of forcing the introduction of non-determinism in programming languages and making them available to the user.*

The redesigning involves being careful about "ownership" of resources and making sure that independent outputs are associated with truly independent devices, windows or other services. Furthermore, based on the same principles, Serrarens [112] built `BriX`, a deterministic concurrent windowing system on top of `ghc` compiler, as this is explained in Section 5.5.2.

Moreover, the issues raised by such a declarative form of concurrency can be of interest in their own right, beyond functional programming and can be adopted in any language or `I/O` model.

### 5.3.4 Implementation

To build deterministic concurrent systems, two requirements need to be met.

*The first requirement is to provide primitives that spawn truly independent threads of execution to correspond to independent demands.*

The independence of threads ensures that distinct threads cannot mutate the same state, since their states are disjoint. This is in contrast with the approach taken in Concurrent Haskell where the `forkIO` primitive allows parent and child threads to mutate the same state which immediately introduces non-determinism, e.g. two threads can access the same file simultaneously.

*The second requirement*[5] *for deterministic concurrency is that determinism needs to be enforced by the design of concurrent systems so that the independence of threads is further preserved; different threads should be addressed to independent devices or services, thus non-deterministic effects cannot arise.*

If two threads go to the same service, say a file manager, which is capable of non-deterministic merge, then the program could produce non-deterministic effects, so it is a designer's obligation to avoid this. In fact, the restriction that distinct threads should be associated with distinct devices has been already adopted in the design of web browsers. In a web browser, one can click only on one link. If however, two links need to operate concurrently, a new web browser is spawned so that each link operates on its own browser.

In Section 5.5, sample deterministic designs [51] for a file manager, a window system and a communications mechanism are outlined.

Again it should be pointed out that deterministic primitives together with deterministic design are expressive enough for coping with single-user aspects of concurrency.

## 5.4 Deterministic Primitives

Deterministic concurrency was first introduced in Haskell 1.2, based on the stream `I/O` model. One of the main purposes of this thesis is to introduce deterministic concurrency into Haskell 1.4 which is based on monads. The next sections describe the two approaches.

### 5.4.1 Deterministic Concurrency in Haskell 1.2

The first deterministic concurrency framework [51] was initially introduced in Haskell 1.2 on top of the `Dialogues` [90] `I/O` model, based on laziness. A sequential Haskell program produces as output a single stream of `I/O` requests

[5] As it can be seen in Chapter 7, the novelty of this thesis in the theory of deterministic concurrency is that it goes beyong good design practise by introducing a deterministic form of communications based on timestamping of messages from different sources.

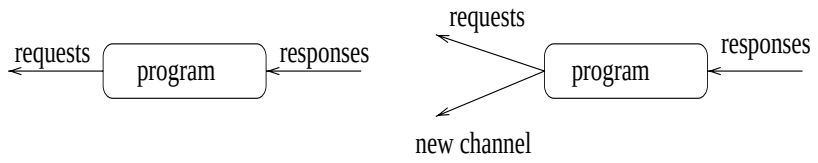



Figure 5.3: Deterministic Concurrency Model

of type `Request` and accepts as input a single stream of corresponding responses of type `Response`.

```
data Request = ReadChan Name | AppendChan Name Stream | ...
data Response = Str Stream | Success | ...
```

In this sequential `I/O` model, the only output request for channels is `AppendChan chan cs`, where `chan` is a channel name and `cs` is a string to be sent to it. This command evaluates the string `cs` in a hyperstrict way and sends it to the channel before the next request. Separate output channels have to be merged into a single request stream with the interleaving being determined explicitly by the program. However, multiple independent output channels cannot be created by `AppendChan`.

To add concurrency on top of this `I/O` model, Holyer and Carter [51] added a new request `WriteChan` to act as a forking mechanism for creating a new channel.

```
data Request = ... | WriteChan Name Stream | ...
```

An expression `WriteChan chan cs` starts up a new thread of execution by adding a new channel to evaluate the stream `cs` independently from the main sequence (rather than evaluate the stream itself), before moving to the next request. In keeping with the deterministic concurrency model, the outside world is regarded as responsible for demanding items on the multiple output streams at will and the program must respond to these demands. The initial thread of execution evaluates and obeys all the requests. When there are no more requests, the main thread terminates. The program though does not terminate, since other threads may still be running; it terminates when all threads terminate. Again any concurrent design should be careful so that distinct threads are associated with distinct devices so that non-deterministic effects do not arise.

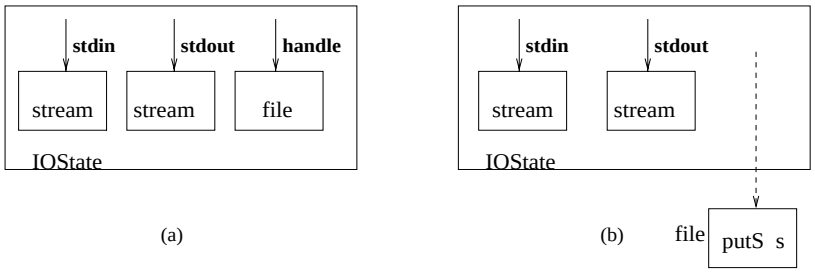



Figure 5.4: (a) State Splitting. (b) State Forking

### 5.4.2 Deterministic Concurrency in Haskell 1.4

Since the monadic `I/O` model of Haskell 1.4 is state based, preserving the independence of demands implies that distinct threads should be restricted so that they act on a different part of the state, which is disjoint from the substates other threads act upon. Thus, non-deterministic effects that arise when threads mutate the same state are avoided. To achieve this, the underlying `I/O` mechanism should support *state splitting* [55]. State splitting is a mechanism which assigns each new state thread[6] part of the state, a *substate*, disjoint from other substates, to act upon. Under this scheme, the state can be visualised as consisting of components, one for each facility mentioned in the program, e.g. a file, a window etc, see also Figure 5.4(a). These components must be independent of each other; actions which involve one component must not affect any other in the same program. Creating a new substate consists in adding a new component to the parent state which is independent from other substates.

Upon such a state splitting mechanism, deterministic concurrency can be built by introducing a state forking one. Similarly, a state forking mechanism carries out an action on a substate of the main state while leaving the remainder of the main state undisturbed. However, in contrast to state splitting, a state forking mechanism creates a separate execution thread, i.e. a new demand to drive an action by evaluating the final substate it produces. This new thread represents an additional demand to any other demands on the program and executes independently of and concurrently with the subsequent actions on the

[6] In monadic `I/O` which is state based, a *state thread* denotes any program fragment or facility that requires use of updatable state, e.g. `I/O` sequences or incremental arrays.

main state, as illustrated in Figure 5.4(b).

In order to incorporate such a state splitting mechanism into Haskell 1.4 [5], the underlying monadic framework needs to be able to provide both facilities for creating independent substates and to allow actions on them. To achieve this, the monadic framework [69], presented in Chapter 4, has been adapted so that it enables the creation of independent state components and allows actions upon them. This gave a new monadic framework, which has been incorporated in the Brisk Compiler and is presented in Chapter 6.

## 5.5 Deterministic Design

The next sections discuss the design of a complete single-user operating system environment built out of functional components as suggested by Holyer and Carter [51]. Such an environment can be built on top of a procedural operating system and its facilities are regarded as support to a functional language rather than facilities which programmers can access directly. This disallows non-determinism to appear at a user level as `I/O` and interfacing problems do not create any user-visible non-determinism.

### 5.5.1 Design for a Purely Deterministic File Manager

The design of a deterministic file manager is presented in this section. This has been initially presented by Holyer and Carter [51] and is adapted for Haskell 1.4. This design assumes that a window system is available in which independent text windows, each with its own input and output streams can be created using a fork primitive `window` as follows:

```
window :: IO a -> IO a
```

This takes as argument an `I/O` action and creates a separate window and a separate thread of control to execute this action. In this way, it is guaranteed that each action is associated with its own independent window. It is also assumed that the only means for performing `I/O` are via standard input `stdin` and standard output `stdout`.

The process and file manager as a whole are considered as a monolithic functional program with processes implemented as threads accessing a single

shared heap. The main program thread creates a window and a new thread for each process.

```
main :: IO ()
main = window processA >> window processB where


ProcessA, processB :: IO ()
processA = ...
processB = ...
```

The main thread then terminates, leaving the two new threads executing independently in separate windows. Since, each thread is created either by creating a new channel (as in Haskell 1.2 [51]) or a new thread that splits off part of the state (as in Haskell 1.4, see Chapter 6), distinct threads are guaranteed to be independent. This ensures that the state of each window is independent of the states of other windows.

To further keep the system deterministic, the following design issues need to be applied so that non-determinism which arises as a consequence of contention does not arise.

- The manager is a process separate from user processes. It provides built-in commands to examine and manipulate file names and directories.
- The manager supports commands for running processes. A user process runs in its own window; it never sends output directly to the manager window.
- The manager has access to file names and directories; user processes don't. User processes only display or alter file contents whereas the manager doesn't.
- User processes are given contents and window input as arguments and return new file contents and window outputs as results. They cannot access files dynamically.
- The manager keeps old versions of files and provides built-in commands to restore them, thus enabling undoing.

This design ensures that several contention problems, such as contention for files, contention for output or non-determinism as a result of interruption are avoided.

Contention for files is avoided because two processes cannot attempt simultaneously to update the same file. When the first process starts up, the manager replaces the file contents with an unevaluated expression representing the new contents which will eventually be returned by the process. When the second process starts up, it is passed the unevaluated expression as an argument and has to wait until the first one finishes.

Contention for output is also avoided, since each process has its own window. Thus, the problem of two processes sending their output to the same window does not arise.

To avoid non-deterministic effects that arise when interrupt signals are sent, an interrupt signal is not considered as a signal to the process, since its has to be non-deterministically merged with other inputs. Instead, an interrupt signal is assumed to be addressed to the file manager. Since the file manager keeps old versions of files, it reverses the contents of the file to a previous version. Additionally, a synchronisation mechanism can be added for saving work up to the point of interruption. Further issues where deterministic design needs to be enforced such as dynamic access to files or how files are committed to permanent storage are illustrated by Holyer and Carter in [51]. A safe undoing facility is further discussed by Pehlivan and Holyer [92]

This file manager, although it might seem that it offers a restricted functionality, it offers enforced security in return. Take as an example, the situation where Java applets are not able to read or write to files in the host computer, load and run programs or communicate with the local server. The file manager presented here, could allow an applet to communicate with files or run programs in the host computer as it prohibits any unrestricted access.

### 5.5.2 Design for a Window System

Many window systems, Fudgets [21], Gadgets [88], Haggis [35] have been designed for Haskell. The Fudgets system is a sequential window interface, Haggis and Gadgets are concurrent. The sequential approach, although inherently awk-

ward, is deterministic, whereas concurrent approaches exhibit non-deterministic behaviour due to the non-determinism introduced by concurrency. The overview of these GUIs is presented in Section 5.7.1.

Deterministic concurrency offers a compromise in expressiveness between the two approaches. The loss of expressiveness with respect to other concurrent windowing systems is outweigh by the predictable behaviour it offers. Based on the principles of deterministic concurrency, a purely deterministic windowing system, BriX [112] has been built on top of the ghc compiler and its design is briefly outlined here.

The basic assumption for building functional GUIs is that a low level window server such as the X Window System [89] is available so that a high level window system can be build on top of it. The window server provides windows which are "nearly" independent of each other. The challenge in a purely functional language is to provide an interface where this independence is complete. In keeping with the deterministic concurrency restriction, where demands are associated with distinct devices, a separate thread is associated with each output stream of graphics commands for each window. To control also subwindows independently, each window is responsible for its children but cannot affect other windows. A basic window library such as Xlib is represented in a functional language as an I/O action:

```
xOpenDisplay :: String -> X a -> IO a
xCreateSimpleWindow :: ... -> X a -> X a
xCreateWindow :: ... -> X Wid
xMoveWindow :: Int -> Int -> Wid -> X ()
xDrawLine :: Int -> Int -> Int -> Int -> X ()
```

Functions xOpenDisplay and xCreateSimpleWindow are both forking actions, i.e. they create a new thread to evaluate action X a. Function xOpenDisplay opens a connection to a given workstation display, to carry action X a. Function xCreatSimpleWindow creates a new window to carry action X a independently of the parent window. Function xOpenWindow creates a new subwindow and returns its identifier Wid to its parent without creating a new thread. Control actions from the parent to the child such as xMoveWindow can be carried out with the child window identifier acting as a reference to the child window substate.

Moreover, a parent can execute an action on its child concurrently with any other actions on its own window. As already mentioned, `BriX` has been implemented on top of the `ghc` compiler where processes mutate all the same state as it does not support state splitting. Thus, extra care has been taken so that distinct threads be kept independent.

Furthermore, control actions such as `xMoveWindow` have been made independent of the graphics appearing on the window so that the parent can carry them out on its children while graphics are appearing on the children.

Actions such as `xDrawLine` are carried out individually by each window, on its own canvas without affecting any other window.

Event handling represents also a potential source of non-determinism. The `X` system provides a single stream of events for all the windows containing information either referring to individual windows or information about relative timings of events concerning more than one windows. If each window was allowed to filter and see only events related directly to it, the relative timing information would be lost and non-deterministic merging should be employed. Thus, a mechanism is needed to associate events with individual windows without losing the relative timing information between more than one windows. In `BriX` an action `getEvent` has been provided which can be called by any window:

```
getEvent :: X Event
```

This acts as a central mechanism which allows a window to be informed of events related not only to itself but also to its subwindow hierarchy (relative timing information). For example, if a menu is a window with a subwindow for each choice, then the event handler allows the main menu window to know all the relative information concerning the choices.

Also low-level events to resize windows have been included together with user input in event streams in a correctly synchronised way since they are a direct effect of user actions.

### 5.5.3 Design for a Communications Mechanism

Holyer and Carter [51] sketched also a communications mechanism in Haskell 1.2 based on deterministic concurrency. Deterministic concurrency offers a sin-

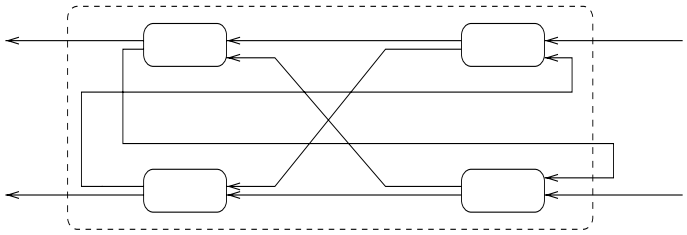


Figure 5.5: Network of Processes as a Single Process

gle heap model with a number of concurrent, semantically transparent threads, each corresponding to an independent demand, that do not share any state. Moreover, this system can be split up into a number of separate processes, each with its own heap and distributed across a number of processors, see also Figure 5.5. Additionally, the collection of processes can cooperate with procedural processes forming a system where each process can either be a concurrent functional program or a procedural program protected from being misused by a functional wrapper. These processes can communicate in two ways:

- via conventional communication channels;
- via specific protocols arranged so that the entire collection of processes is equivalent to a single heap program.

The second point is implied by the fact that deterministic concurrency preserves the declarative semantics of programs. These protocols restrict the use of conventional communications channels by replacing them with stream processing functions (available in Haskell 1.2) with equivalent functionality. In other words, deterministic concurrency allows such transformations that an arbitrary network of processes connected by conventional communications channels is equivalent to a single distributed heap, the only difference being that a network has more internal parallelism. This equivalence is not obtained by any of the conventional process algebras like `CSP` [49, 48], `CCS` [82] or $\pi$-calculus [83]. This is the result of bringing communication into the language and making it referentially transparent. As we shall see in Chapter 10, this is the key issue to allow for arbitrary migration of computation without any change in the meaning of a program.

Now, we analyse the protocols that offer this equivalence in Haskell 1.2. Channels either carry data streams or request and response streams. A channel which carries a data stream is equivalent to a protocol that passes values from one process to another. A channel which carries a request or response stream is equivalent to a protocol that sets up a new channel connection or in other words passing a lazy list value from one process to another.

Also a protocol that creates a new process is equivalent to a function which takes streams as arguments and returns streams as results. Such a facility to create new processes together with the `writeChan` facility described above to create channels makes the network completely dynamic.

The design of such a communications mechanism has been the starting point in building a purely functional distributed system, see also Chapters 10, 11. Within the semantic framework that is being developed for deterministic concurrency, distribution is almost an orthogonal issue. Building distributed systems on top of the deterministic concurrency model, enables programs to be distributed without any change to the "meaning" of the programs -the only observable difference being the rate at which the overall system generates the results, which also offers a framework for mobile computation.

### 5.5.4 A Development Environment

The single-user aspects of an operating system discussed so far namely a file manager, a windowing system and a communications mechanism suggest that deterministic concurrency is expressive enough to implement a complete single-user multi-tasking working environment.

Moreover, such an environment can be viewed as a single persistent program, where compiling and running a new module is equivalent to creating new functions and data types dynamically and then loading them up dynamically into the already running program. Of course this requires dynamic loading and dynamic types. Dynamic loading has been a crucial factor in the design of the Brisk Machine [54], which is one of the main components of this thesis and is fully described in Chapters 8 and 9. Dynamic types have been also described by Dornan in his PhD thesis [31].

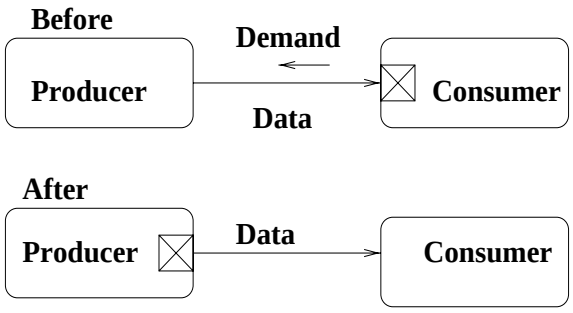



Figure 5.6: Demand Driven Buffered Protocols

## 5.6 Synchronisation

The traditional aspect of concurrency is tightly associated with shared state. Thus, synchronisation is necessary to ensure that the states of all the processes in a concurrent program remain consistent. On the other hand, deterministic concurrency is not based on shared state. Threads are independent of each other since they are created so that that they act on different parts of the state. Additionally, careful deterministic design of concurrent systems imposes such restrictions, so that threads be kept independent of each other. This implies that deterministic concurrency, as it is not based on shared state, does not require any facilities for mutual exclusion.

Traditional buffer protocols also have a different flavour under the deterministic concurrency approach. A simple unbuffered protocol under a demand driven point of view can be described as follows: when the consumer requires a value from the producer, it creates a "demand token" for the next item from the producer and is suspended, see also Figure 5.6. The producer is woken, evaluates the item, sends it and suspends itself again. The arrival of the item allows the consumer to continue. A demand driven buffered protocol also is one in which a number of items are demanded before they are needed, so that the producer evaluates them speculatively.

## 5.7 Communication

In paragraph 5.5.3, the design of communications mechanism based on deterministic concurrency has been sketched. This mechanism allows processes to communicate in a novel way, via specific protocols arranged so that the entire collection of processes is equivalent to a single heap program. These protocols restrict the use of conventional communications channels by replacing them with stream processing functions (available in Haskell 1.2) with equivalent functionality. In other words, deterministic concurrency, as it has been described by Holyer and Carter [51, 23] assumes a number of concurrent threads which do not communicate either via shared variables or via channels, since that would require some amount of non-determinism. They can communicate only via shared values (not stateful ones), including streams which they take as arguments or produce as results. The laziness of these streams makes this form of communication more effective than one might at first think, as it has been proven expressive enough to implement a single-user multi-tasking system and a deterministic graphical user interface.

On the other hand, this thesis raises an objection towards this form of communication. Although such a form of concurrency offers important theoretical insights, it is restrictive. It imposes processes to be arranged in a tree-like structure, see also Figure 5.7(a), where each process has only one input.

As a consequence, it cannot be used as a substrate for building any multi-user applications, such as a multi-user operating system. In such concurrent systems, processes are usually arranged in cycles and each process might have more than one input, see also Figure 5.7(b). This immediately introduces the problem of merging streams of information, which introduces non-determinism. In a more theoretical basis, deterministic concurrency does not provide a solution to any multi-user aspect of concurrency such as the dining philosophers problem or the readers and writers problem.

Moreover, even in single-user aspects of concurrency, this form of communication in deterministic concurrency cannot cope with current trends in concurrent program designs. Such designs, particularly for `GUIs`, tend to use a freer style of programming in which arbitrary networks of components, communicating via channels, are set up. This style encourages good modularity of compo-

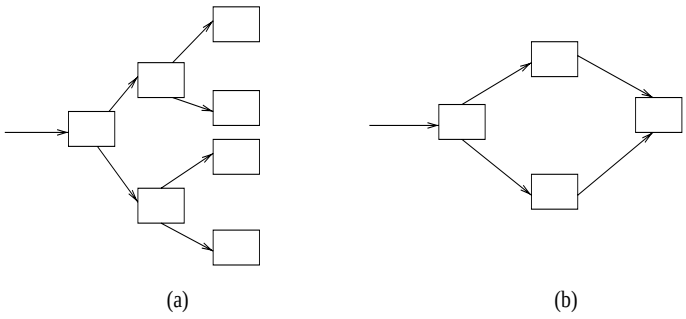



Figure 5.7:

nents in an object-oriented style. To achieve this, event handling is organised in terms of an arbitrary network of processes which communicate asynchronously by sending messages to each other via channels and an arbitrary communication network links them. Many purely functional `GUI` libraries such as Haggis [35] and Gadgets [88] have been designed according to this philosophy. In such libraries, the graphics network is separated from the event network. Then the event network is made arbitrary by naming communication channels. This amounts to an asynchronous communication message passing model, but on the face of it, such unrestrained communication immediately re-introduces non-deterministic merging of message streams. Having processes connected by an arbitrary communication network and communicate asynchronously via channels is highly non-deterministic, since it implies the use of a non-deterministic merge. To complement this picture, a brief overview of some of the most important purely functional `GUIs` is given in Section 5.7.1.

To overcome the above mentioned problems, this thesis investigates also how to make deterministic concurrency more expressive by offering a form of *deterministic communications*. This is achieved by attaching timing information in form of timestamps added to streams. These timestamps impose a linear order on all messages, allowing them to be merged in a consistent temporal order. The timestamping is not based on real wall-clock time [67], because maintaining and distributing a universal notion of time is difficult and it would not give a satisfactorily complete notion of determinism. To be suitable for a distributed setting where no shared global clock is available, timestamping is based on a logical notion of time. This logical notion of time allows messages from different

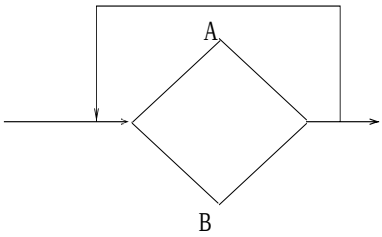



Figure 5.8: Processing in Fudgets

sources to be merged in a deterministic way into a single stream, whereas it affects concurrency to a minimum. This is fully analysed in Chapter 7.

### 5.7.1 Overview of Purely Functional GUIs

**Fudgets** The Fudgets [21] system, has been the first purely functional message passing `GUI` built on top of the sequential Haskell compiler `hbc`. Components interact with each other via message passing and combinators are provided to glue components into bigger programs. Additionally, off-the-shelf components are offered that can be used to create programs with graphical user interfaces.

Fudgets provide a restricted message passing approach. Each Fudget has only a single input and a single output through which to pass messages. Although message passing allows the graphics layout to be independent from the event network, the above restriction imposes that the graphics layout follows the event network. This is restrictive and implies that the structure of the main program is forced to follow the structure of the process network. As a consequence, Fudgets cannot provide dynamically changing networks. The network structure has to remain the same as when it was first constructed. This is partly because of message routing; as a request from a Fudget (thread) goes through the network, a route is created which is used to send back the reply. This requires, that the network structure must not have changed by the time the reply is sent, otherwise it won't reach the right place in the network. If there is a large loop in a program, the output from the program as a whole is sent back to the beginning and merged with the main input, see also Figure 5.8. This means that all processing triggered by one event must cease before the next event can be read in and processed.

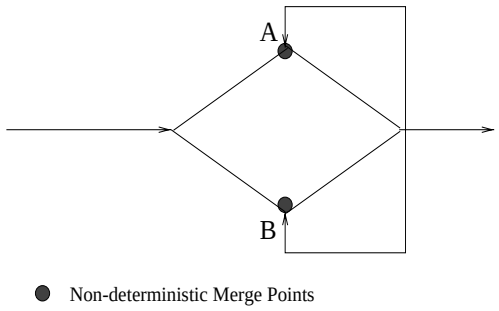



Figure 5.9: Processing in Gadgets and Haggis

On the other hand, the restriction that there is exactly one input channel and one output channel per thread and that threads have only a small fixed set of carefully constructed combining forms offers an important advantage. Fudgets have a form of communication based on merging which is deterministic. The set of combining forms can be added on top of a sequential Haskell and allow for deterministic merging.

**Gadgets** Gadgets is another purely functional graphical user interface that overcame the problems caused by Fudgets in restricting the network structure to be fixed. They have fully dynamic networks which support process creation and changes in channel connections in a non-deterministic concurrent setting, with arbitrary non-deterministic merging, see also Figure 5.9.

**Haggis** The Haggis system offers a user interface which builds on top of the monadic `I/O` principles of Concurrent Haskell. Each component is a virtual `I/O` device, i.e. it is identified by a handle and is accessed through monadic actions like conventional `I/O` devices. Therefore, it possesses all the virtues of the monadic `I/O` such as composability, extensibility. Process creation and process communication is based on Concurrent Haskell described in Section 5.2.6. Again, non-deterministic merge has been introduced, see also Figure 5.9, but as already explained, non-determinism is pushed out of the functional program to the `I/O` state leaving the functional program referentially transparent.

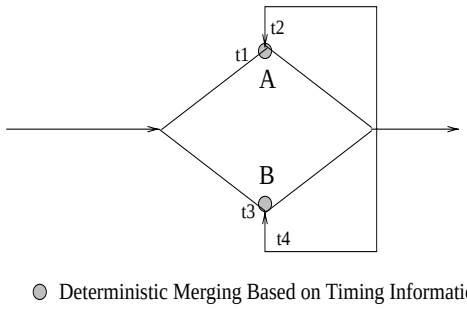



Figure 5.10: Deterministic Merging

### 5.7.2 Deterministic Merge

In Chapter 7, an approach based on timestamping events from different sources is presented, see also Figure 5.10. The timestamping is based on an hierarchical notion of time. The use of timestamps on communication events between threads together with synchronisation messages that broadcast state information ensures, in a transparent way, that all merging of message streams in a program is deterministic, without any unnecessary delays. The timestamping approach also can be used as a mechanism for incorporating multi-user facilities in a way which reduces their inherent non-determinism to a minimum.

In contrast to the approach taken in `RUTH`, described in Section 5.2.3, when a process has more than one channels it is prevented from accepting a message until it has been determined that no earlier messages can arrive in any of the channels.

Adding safe facilities for creating threads and channels leads to a system corresponding to a deterministic subset of a concurrency model such as the $\pi$-calculus [83]. In this subset there may be internal non-deterministic evolution but the observational output from each independent process (source of demand) is weakly bisimilar to a deterministic description acting over the relevant inputs.

## 5.8 Deterministic Concurrency vs Concurrent Haskell

Concurrent Haskell [36] is based on the traditional concurrency model. This causes programs to exhibit non-deterministic behaviour which is unwanted by programmers and users. Non-determinism is introduced during thread creation, as concurrent threads mutate the same state. Additionally, the primitive `mergeIO` merges non-deterministically communication events during process communication. Non-determinism has been hidden away from the functional program into `I/O` state and thus, programs remain semantically referentially transparent. Despite the adverse effects of non-determinism, Concurrent Haskell offered a successful approach for single self-contained programs within conventional procedural operating systems.

However, this approach requires a non-deterministic operating system to handle the amount of non-determinism that comes out of the functional program. Since all existing operating systems exhibit non-deterministic behaviour, this has not been considered a problem. On the other hand, Concurrent Haskell's non-deterministic primitives cannot be added on top of the purely declarative operating system suggested by Carter [23] as it is clear that such an operating system cannot cope with the amount of non-determinism coming out of the functional program.

Moreover, this approach is not sufficient in the case of distributed programs built from communicating components, where a program must be deterministic as a whole, not just each of its parts. This requires bringing communication inside the language and making it inherently referentially transparent, which is a key issue to allow for arbitrary migration of computation without any change in the meaning of a program.

On the other hand, deterministic concurrency, as already described, is a purely declarative form of concurrency which does not allow non-determinism to appear at a user-level. Although it provides a restricted concurrency model, the gains from such a form of concurrency outweigh the restrictions.

Another important issue is that, deterministic concurrency, when used as a basis for a distributed model, allows for a novel form of computational mobility

in a demand driven style since it exhibits performance flexibility, i.e. it maximises the scope for evaluation independent of location and can naturally express distribution, migration and mobility. This is fully discussed in Chapter 10.

# Chapter 6

# Concurrent Monadic Interfacing

This chapter presents the Brisk monadic framework, in which the usual monadic style of interfacing is adapted to accommodate a deterministic form of concurrency. Its main innovation is to allow actions on state components. This is a key issue which enables *state splitting*, a technique which assigns to each new state thread a part of the state, a *substate*, to act upon. Distinct concurrent threads are restricted to access disjoint substates.

Upon such a state splitting mechanism, a state forking one has been added. This creates a new execution thread, i.e. a new demand to drive an action by evaluating the final substate it produces. The resulting system acts as a basis for offering a purely functional form of concurrency, extending the expressiveness of functional languages without spoiling the semantics by introducing non-determinism.

Such an extended monadic framework offers two additional advantages. It provides a modularised approach to state types, where each state type and primitive operations can be defined in its own module. Furthermore, it improves some existing language features by making them deterministic, e.g. reading characters lazily from a file without unexpected effects. We also provide a lazy version of the standard Haskell function `writeFile` for writing characters lazily to a file.

## 6.1 Introduction

In Chapter 5, a new model of concurrency, *deterministic concurrency*, has been introduced. Again, deterministic concurrency [23, 51] is a purely declarative approach to concurrency which extends the demand driven evaluation model of functional languages into one with multiple independent demands. The independence of demands can only be guaranteed by restricting distinct threads so that each acts on a different portion of the state, a *substate*, which is disjoint from the substates other threads act upon. Thus, non-deterministic effects that arise when threads mutate the same state are avoided. To achieve this, we use the technique of *state splitting*. State splitting is a mechanism which assigns at creation of each new state thread a substate, disjoint from other substates, to act upon. This approach enables a concurrent setting where several independent threads access the outside world, i.e. the `I/O` state of the program. Under such a deterministic approach to concurrency, a processor can run one or more `I/O` performing operations without exhibiting any user-visible non-determinism.

In order to incorporate such a state splitting mechanism for supporting deterministic concurrency into Haskell 1.4, the underlying monadic framework needs to be extended with the ability to provide independent substates and to allow actions on such state components. In this chapter, the Brisk monadic framework is presented, adopted in the Brisk compiler. This framework retains all the state-based facilities provided in the existing one. Additionally it is extended so that it enables the creation of independent state components and actions upon them. Such an extended monadic framework offers two additional advantages. It provides a modularised approach to state types, where each state type and primitive operations can be defined in its own module. Furthermore, it improves some existing language features by making them deterministic, e.g. reading characters lazily from a file without unexpected effects. We also provide a lazy version of the standard Haskell function `writeFile` for writing characters lazily to a file.

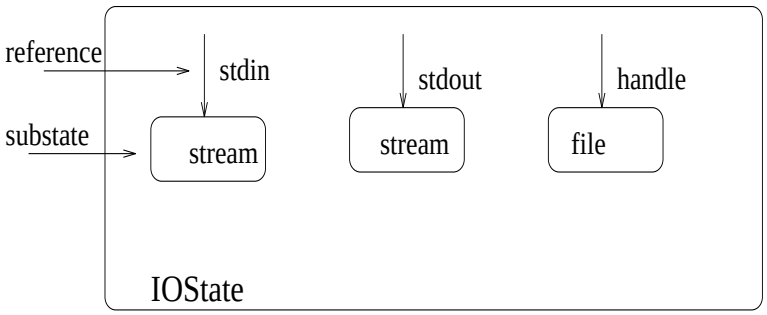



Figure 6.1: State Splitting

## 6.2 The Brisk Monadic Framework

The main issue that has permeated the design of the Brisk monadic framework is to allow states to have independent state components or substates, with separate threads acting on those substates, possibly concurrently. Specific functions are provided to create and act on substates. Each substate is associated with a reference and every access to a substate happens via this reference, as in Figure 6.1. This implies that internal state types such as arrays are separated from the reference mechanism, as described by Launchbury and Peyton Jones [69] and they no longer need to be accessed via references, unless they happen to be substates of some other state. Thus, in the Brisk monadic framework there is no longer a distinction between internal (i.e. arrays) and external (i.e. `I/O`) state types. Instead, each facility has its own concrete type, so that both internal and external facilities are treated uniformly. Moreover, each state type can be defined separately in its own module. Similarly, a group of related foreign procedure calls can be encapsulated in its own module by defining a suitable state type and presenting the procedures as suitable actions.

In the next sections, the Brisk monadic framework is outlined and all the facilities that derive from it are illustrated. First, we show how all the facilities mentioned in the `ghc` monadic framework [69] are reproduced in the Brisk one and then we present additional features that derive from our approach.

### 6.2.1 Generic Primitives

In the Brisk framework, any stateful computation is expressed via the type:

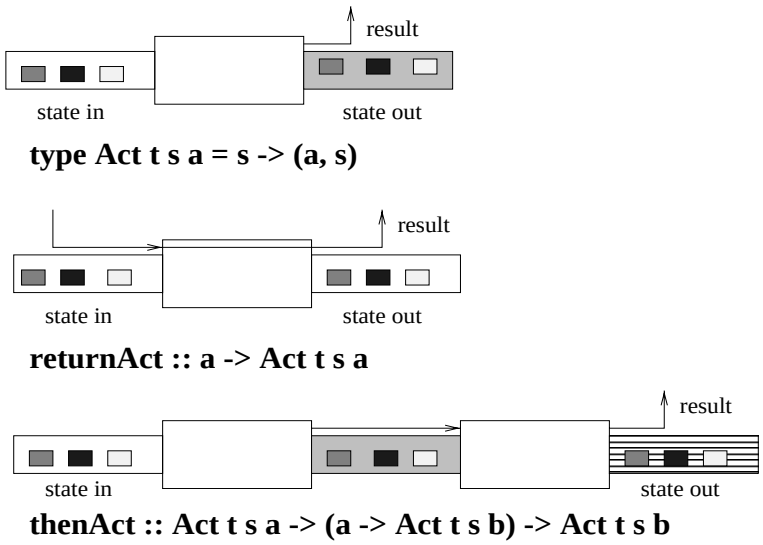



Figure 6.2: Monadic Operators

```
Act t s a = s -> (s, a)
```

which denotes a state transformer that transforms a component-state and returns a new component-state and a result of type `a`, together with operations `returnAct` and `thenAct` for accessing it:

```
returnAct :: a -> Act t s a
thenAct :: Act t s a -> (a -> Act t s b) -> Act t s b
```

The type `Act t s a` is very similar to `ST s a`, but we have changed the name to avoid confusion between the two. As with `ST`, `Act` is made abstract for protection. An action of type `Act t s a` transforms a state of type `s` (consisted of components) and delivers a result of type `a` together with a new state of type `s` (consisted of components), as can be seen in Figure 6.2. However, `s` will now normally be instantiated to a specific state type. The second purpose of `s` in `ST s a`, namely to act as a tag identifying a specific state thread, is separated out into the extra type variable `t`, which is never instantiated. Function `returnAct` returns a value without affecting the state and function `thenAct` applies the result of the computation `Act t s a` to the function `a -> Act t s b` resulting computation `Act t s b`.

As an example, we will show later that `I/O` facilities can be defined essentially as in Haskell with:

```
type IO t a = Act t RealWorld a
```

The `IO` type now has a tag `t` on it. It is possible to make this tagging less visible to the programmer, but we do not pursue that here. We will see later that specialised state-splitting facilities can be provided for safe handling of concurrent `I/O` threads.

For encapsulating stateful computations into non-stateful ones, the `run` function has been provided:

```
run :: s -> (forall t. Act t s a) -> a
```

`run` takes an initial value for the state and an action to perform. It carries out the given action on the initial state, discards the final state and returns the result. As with `runST`, the tag variable `t` identifies a particular state thread at compile-time. Forcing the action to be polymorphic in `t` encapsulates the state thread, preventing references from escaping from their scope or being used in the wrong state. To prevent `run` from being used in conjunction with `IO`, the `IO` module does not export any useful state value which can be used as an initial state.

### 6.2.2 References

One of the main changes in the Brisk monadic framework concerns the philosophy of references. A references is a pointer to a substate, see also Figure 6.1.

```
type Ref t u s = ...
```

References are treated as being independent of any specific substate type. The reference type has two tags. The first refers to the main state and ensures that the reference is used in the state where it was created. The second tag refers to the substate and ensures that the reference can only be used while the substate actually exists.

The generic facilities for handling substates and references are:

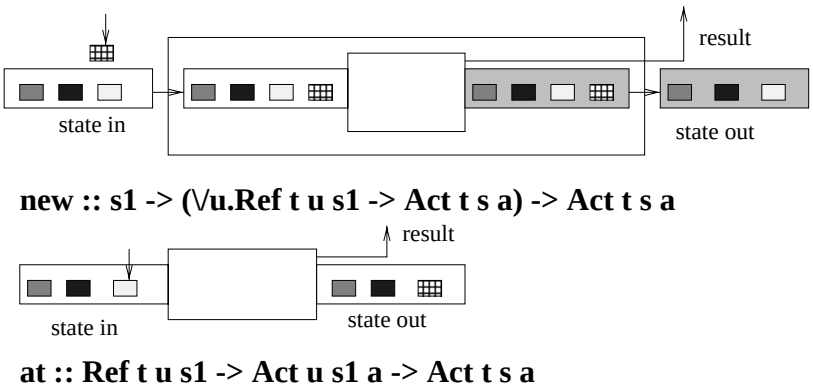



Figure 6.3: Operators for Handling Substates

```
new :: s1 -> (forall u. Ref t u s1 -> Act t s a) -> Act t s a
at :: Ref t u s1 -> Act u s1 a -> Act t s a
```

The `new` function is introduced to implement state splitting. It creates a new substate of type `s1` with a given initial value. It does this for any desired type `s` of main state. It also creates a reference to the new substate, as can be seen in Figure 6.3. The second argument to `new` is a function which takes the new reference and performs the actions on the new extended main state. This contrasts with definitions such as the one for `openFile` where a reference is returned directly and the main action sequence continues. However, in a context where there may be substates and subsubstates and so on, this encapsulation of the actions on the extended state is needed to provide a scope for the new tag variable `u` and to protect references from misuse.

The function `at`, see also Figure 6.3, carries out an action on a substate. The call `at r f` performs action `f` on the substate corresponding to the reference `r`. This is regarded as an action on the main state in which the rest of the state (the main state value and the other substates) is unaltered. In the main state, the substate accessible via `r` is replaced by the final state after the primitive actions in `f` have been performed. The actions in `f` are performed lazily, i.e. driven by demand on its result and may be interleaved with later actions in the main thread. This assumes that the substate is independent of the main state so that the order of interleaving does not matter. Consider a sequence of actions:

```
a1 >> at r a2 >> a3 >> at r a4 >> a5 ...
```

As in Haskell, we use overloaded functions `>>=` and `>>` and `return` for monadic operations, in this case referring to `thenAct` etc. The laziness of `thenAct` means that the actions in the sequence are driven only by the demand for results. The usual convention that primitive actions should be strict in the state ensures that the actions happen in sequence. However, main state actions such as `a3` are not strict in the substates of the main state. The action `at r a2` replaces the substate referred to by `r` by an unevaluated expression; it is the next action `a4` on the substate which causes `a2` to be performed. Thus, the sequence of actions `a1`, `a3`, ... on the main state and the sequence `a2`, `a4`, ... on the substate, may be executed independently. We are assuming here that the substate is independent of the main state, so that the order of interleaving does not matter. The laziness of the `thenAct` operator allows concurrency effects to be obtained in this way.

It is possible that a main state action such as `a5` may cause the evaluation of the result of some previous action on a substate such as `a4`. In this case, there is effectively a synchronisation point, forcing all previous actions on the substate to be completed before the main state action is performed.

Now, each individual state type and its primitive actions can be defined separately in its own module. It is easy to add new state types, e.g. out of values or procedural interfaces. Furthermore, this modularised point of view can be used for providing separate interfaces to separate `C` libraries, instead of using a primitive `ccall`.

### 6.2.3 Semantics

The semantics of our proposed system does not require anything beyond normal lazy functional semantics. In particular, although state splitting is designed to work alongside concurrency, it is independent of concurrency and can be described in sequential terms. Thus, instead of presenting the semantics directly with a mathematical formalism, we choose to present it using a prototype implementation in Haskell. For the prototype implementation, we use hugs-98 with the (-98) extension which supports 2nd order polymorphism, a common extension to Haskell at the time of writing. The principal difficulty in producing the prototype is not in capturing the correct semantics, but rather in getting

around Haskell's type system. When substates are added to a state, it would seem that the type of the main state needs to change, but the type changes are too dynamic to be captured by the usual polymorphism or overloading mechanisms.[1]

In the prototype implementation presented here, the state type `State s` is polymorphic and describes a state with a value of type `s` and a list of substates. To get around Haskell's type system, substates are not described using type `State` but instead are given the substate type `Sub`. This is not polymorphic, but is a union of specific instantiations of the `State` type. Each type which can form a substate is made an instance of the `Substate` class, with functions `sub` and `unsub` which convert the generic state representation to and from the specific substate representation.

```
data State s = S s [Sub]

data Sub = SI Int [Sub] | SA (Array Int Int) [Sub] |
           SF File [Sub] | SW World [Sub] |
           SS StInt [Sub] | ...

class Substate s where
   sub :: State s -> Sub
   unsub :: Sub -> State s

instance Substate Int where
   sub (S n ss) = SI n ss
   unsub (SI n ss) = S n ss

instance Substate (Array Int Int) where
   sub (S ar ss) = SA ar ss
   unsub (SA ar ss) = S ar ss
```

[1] Claessen, encountered the same difficulty in an attempt to implement in pure Haskell the standard `ST` monad (with polymorphic `newVar`, `writeVar` and `readVar`). In an e-mail at the Haskell mailing list [26] with title "ST in pure Haskell", he also pointed out that existential/universal quantification in types are not enough but also dynamic types and explicit type casts are required to cope with such an experiment.

...

This use of overloading breaks the desired modularisation of state types, but it is only necessary in this prototype. In a real implementation where state facilities are built in, none of the above is necessary. State types do not need the `State` wrapper and substates can be treated directly. In the code below, calls to `sub` and `unsub` can be ignored.

The `Act` type and the generic operations `returnAct`, `thenAct` and `run` are defined in the usual way. The definition of `run` uses `seq` to evaluate the initial state before beginning in order to prevent error values from being used as initial state values. [2] In the compiler, `Act t s a` is a builtin abstract type, together with the three builtin operations, the types of which are shown here.

```
newtype Act t s a = Act (State s -> (a, State s))

returnAct :: a -> Act t s a
returnAct x = Act (\st -> (x, st))

thenAct :: Act t s a -> (a -> Act t s b) -> Act t s b
thenAct (Act f) g =
   Act (\st->let (x,st1) = f st; Act h = g x in h st1)

run :: s -> (forall t . Act t s a) -> a
run v act = seq v (fst (f (S v []))) where Act f = act
```

A reference is implemented as an integer for simplicity, used as an index into the list of substates. The `Ref` type is tagged with `t` and `u` tags respectively, so that actions on both the main state and the substate can be encapsulated safely. The implementation of `new` uses `seq` to prevent error values being used as initial substates, adds the new substate to the end of the list of substates, carries out the actions which use the new reference and then removes the final version of the substate. The `at` function replaces the relevant substate by the result of applying the given action on it. In a real implementation, `Ref t u s`

[2] Preventing error values to be used as initial state types is not essential, but it allows compiler optimisations in which the state type is implicit rather than explicit.

and the two operations would be builtin; the types would be the same, with the omission of the `Substate` restriction.

```
data Ref t u s = R Int


new :: Substate s1 =>
   s1 -> (forall u . Ref t u s1 -> Act t s a) -> Act t s a
new v1 f = Act g where
   g (S v ss) = seq v1 (x, S v' (init ss')) where
      (x, S v' ss') = h (S v (ss ++ [sub (S v1 [])]))
      Act h = f (R (length ss))


at :: Substate s1 => Ref t u s1 -> Act u s1 a -> Act t s a
at (R r) (Act f) = Act g where
   g (S v ss) = (x, st') where
      subst = unsub (ss !! r)
      (x, subst') = f subst
      st' = S v (take r ss ++ [sub subst'] ++ drop (r+1) ss)
```

This completes the generic facilities in the prototype. A number of examples can be demonstrated by extending the prototype.

## 6.3 Examples of State Types

In this section we demonstrate the way in which different state types can be developed in a modular way and discuss some of the issues that arise.

### 6.3.1 Values as States

State types are usually special abstract types representing updatable data structures or interfaces to procedural facilities. However, it is often convenient to carry ordinary values around in a stateful way, e.g. as substates. This can be illustrated by extending the prototype, showing how to treat `Int` as a state type. The `get` and `put` actions get and put an `Int` value held in the `StInt` state type ("stateful Int"), and `inc` increments it.

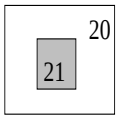



Figure 6.4: An Integer State with an Integer Substate

```
newtype StInt = St Int

get :: Act t StInt Int
get = action f where f (St n) = (n, St n)

put :: Int -> Act t StInt ()
put m = action f where f (St n) = ((), St m)

inc :: Act t StInt ()
inc = get >>= \n -> put (n+1)

action :: (s -> (a,s)) -> Act t s a
action f = Act g where
   g (S v ss) = (x, S v' ss) where (x,v') = f v
```

The function `action` converts an ordinary function of a suitable type into an action on the corresponding state type. It is a convenience function included just for the purposes of the prototype and we will also use it in later examples (it should not be made available in a real implementation because it is unsafe).

The following example, see also Figure 6.4, creates an `StInt` main state and an `StInt` substate of that main state.

```
eg = run (St 20) (new (St 21) (\r ->
   at r inc >> at r get >>= \n ->
   get >>= \m -> return (m+n)))
```

In the above example, function `get` fetches the value of the main state, whereas in order to get the value of the substate the function `at` uses the reference `r` to apply the function `get` to the substate and get its value.

In the prototype, the example is restricted to specific types such as `StInt` in order to insert them individually into the substate union. In the compiler, the facilities shown here can be polymorphic, using `newtype St a = St a`.

Furthermore, for defining an integer state, the `St a` type is used rather than just `a` to prevent illegal access to, e.g. `I/O` states. The `get` action cannot be used during a normal `I/O` sequence because the type of `get` is not compatible with `I/O` actions. At best it has type `Act t (St RealWorld) RealWorld`. It can then only be used as part of an action sequence with state type `St RealWorld` and that sequence must have been started up with `run` or `new`. The lack of any useful values of type `RealWorld` with which to initialise `run` or `new` prevents such a sequence.

As already explained, similarly to the `ghc` compiler [69], rank-2 polymorphic types safely encapsulate actions on the state thread by forcing actions to be polymorphic in a separate type variable `t`. Actions on substates are encapsulated as well by adding an additional polymorphic type variable `u`. The following illegal example illustrates that rank-2 polymorphic types enable the secure encapsulation of state thread accesses:

```
bad = run 0
 (put 41 >>new 0 (\r ->return r) >>= \r1 ->return r1)
```

The second argument to `new`, i.e. the expression `(\r → return r)` is of type `Ref t u s1 → Act t s (Ref t u s1)`. This is disallowed because the tag `u` is not polymorphic, since it appears in both the `Ref` and `Act` types. The second argument to `run` also breaks the rank-2 polymorphic type restriction. Similarly expressions such as the following are illegal and thus references are prevented from being imported into `run` or `new`.

```
... \r -> run (... at r act ...)
... \r -> new (\r1 -> ... at r act ...)
```

Another useful facility due to the modularised approach to state types would be to convert any ordinary stream function

```
f :: [X] -> [Y]
```

into a monadic form

```
([X] -> [Y]) -> Act t s a
```

for some suitable `s`, where items can be both given to `f` and taken from `f` one at a time. Consider for example a stream function `f :: [Int ] -> [Int]` and the following functions `run_stream` and `next` that initialise a monad and operate on it respectively.

```
f :: [Int] -> [Int]


run_stream :: ([Int] -> [Int]) -> Act t s a
next :: Int -> Act t s Int
```

Function `run_stream` initialises the monad by plugging `([Int]` $\rightarrow$ `[Int])` as the state. Function `next` is an operation on the monad that has a stream function embedded in it. It needs to be built-in because it has to do non-functional manipulation. Then, the expression, `f [3,2,5,1] = [3,5,10,11]` can be converted into monadic form as follows:

```
next 3 >>= \n ->  -- (n = 3)
next 2 >>= \n ->  -- (n = 5)
next 5 >>= \n ->  -- (n = 10)
next 1 >>= \n ->  -- (n = 11)
```

This generalises the emulation of `Dialogues` in terms of `IO` as described by Peyton Jones and Wadler [106] and gives a builtin implementation that avoids the space leaks.[3]

```
dialogueToIO :: ([Response] -> [Request]) -> IO t ()
```

### 6.3.2 Arrays

As another example, we show here how to provide incremental arrays. They are described as state types in their own right, not as an array of references as in the `ghc` compiler, as explained in Section 4.2.2. The implementation shown here uses Haskell's monolithic array type `Array i e` as the state type for illustration,

[3] Wadler and Peyton Jones [106] presented a purely functional emulation with space leaks.

where `i` is the index type and `e` is the element type. In practice, a built-in implementation would be provided with a separate state type, supporting update-in-place. A minimal set of facilities might be:

```
getItem :: Int -> Act t (Array Int Int) Int
getItem n = action f where f arr = (arr ! n, arr)

putItem :: Int -> Int -> Act t (Array Int Int) ()
putItem n x = action f where f arr = ((), (arr//[(n,x)]))
```

The `getItem` function finds the array element at a given position and `putItem` updates the element at a given position with a given value. If an array is used as a substate via reference `r`, then its elements are accessed using calls such as `at r (getItem n)` or `at r (putItem n v)`.

Once again, specific index and element types are given for illustration in the prototype. The `getItem` and `putItem` actions can be more general in a real implementation.

### 6.3.3 File Handling

For handling files piecemeal rather than as a whole, a state type `File` is introduced to represent an open file.

```
newtype File = F String

getC :: Act t File Char
getC = action f where f (F (c:cs)) = (c, F cs)

putC :: Char -> Act t File ()
putC c = action f where f (F s) = ((), F (s++[c]))

eof :: Act t File Bool
eof = action f where f (F s) = (null s, F s)

contents :: Act t File String
contents = action f where f (F s) = (s, F s)
```

This is intended to be used as a substate of an input/output state.

### 6.3.4 Input and Output

In this section, we show the implementation of some common `I/O` facilities using the state-splitting ideas described so far. Issues such as error handling are ignored and facilities involving true concurrency are left until later. We treat `I/O` actions as actions on a global `World` state consisting of a number of files, each with a boolean value as an indication whether it is currently open or not. The type `Handle` is regarded as a reference to a `File` substate. This is like the corresponding standard Haskell type, except that in our setting it has two tag variables attached, one representing the main state and the other the `File` substate.

```
newtype World = W [(FilePath, File, Bool)]
type IO t a = Act t World a
type Handle t u = Ref t u File
data IOMode = ReadMode | WriteMode | AppendMode
```

The `openFile` function is one which creates a new `File` substate of the `I/O` state, returning a handle as a reference to it, properly encapsulated. Its implementation ensures that one cannot open a file that does not exist for reading. If a file is opened for writing, its contents are set to the null string.

```
openFile :: FilePath -> IOMode ->
            (forall u . Handle t u -> IO t a) -> IO t a
openFile name mode io = Act g where
 g (S w ss) =
  if isLocked w name then (error "locked", S w ss) else
  (x, S w2 (init ss')) where
  (x, S w1 ss') = h (S (lock w name) (ss ++ [sub (S file [])]))
  S file' [] = unsub (last ss')
  Act h = io (R (length ss))
  file = if (mode == WriteMode) then F "" else getFile w name
  w2 = unlock (putFile w name file') name
```

```
getFile :: World -> FilePath -> File
getFile w@(W ps) f = let i = find f w in thrd3 (ps !! i)

putFile :: World -> FilePath -> File -> World
putFile w@(W ps) f file =
   let i = find f w in
   let (b,_,_) = ps !! i in
   W (take i ps ++ [(b,f,file)] ++ drop (i+1) ps)

lock :: World -> FilePath -> World
lock w@(W ps) f =
   let i = find f w in
   let (b,_,contents) = ps !! i in
   W (take i ps ++ [(True,f,contents)] ++ drop (i+1) ps)

unlock :: World -> FilePath -> World
unlock w@(W ps) f =
   let i = find f w in
   let (b,_,contents) = ps !! i in
   W (take i ps ++ [(False,f,contents)] ++ drop (i+1) ps)

isLocked :: World -> FilePath -> Bool
isLocked w@(W ps) f =
   if not (exists f w) then False else
   fst3 (ps !! (find f w))

find :: FilePath -> World -> Int
find name (W ps) = loop 0 ps where
  loop i [] = error "file not found"
  loop i ((b,n,f):ps) =
   if (n == name) then i else loop (i+1) ps

exists :: FilePath -> World -> Bool
```

```
exists name (W ps) = loop name ps where
 loop name [] = False
 loop name ((b,n,f):ps) =
   if (n == name) then True else loop name ps
```

The `getFile` and `putFile` functions get and put the contents of a file with a given name respectively. The `lock`, `unlock` and `isLocked` functions set, unset and test the boolean flag associated with each file. In this prototype, a file cannot be reopened while already open, even for reading; in a real implementation, multiple readers could be allowed.

The `openFile` function can be thought of as a specialised form of the `new` function. Instead of creating a new substate from nothing, it uses a portion of the main state (a file) to form a new substate. While the substate is in use, the corresponding part of the main state must be locked in whatever way is necessary to ensure that the main state and the substate are independent. For example, the following is not allowed:

```
openFile "data" (\handle -> ... >> openFile "data" ...)
```

When `openFile` returns, the actions on the substate may not necessarily have been completed, since they are driven by a separate demand. However, an action which re-opens the same file after this, acts as a synchronisation mechanism, forcing all the previous actions on the substate to be completed. For example, the following is allowed:

```
openFile "data" (\handle -> ...) >> openFile "data" ...
```

The synchronisation may mean reading the entire contents of a file into memory, for example. In detecting when the same file is re-opened, care has to be taken over filename aliasing. This is where the operating system allows the same physical file to be known by two or more different names, or to be accessed via two or more different routes.

The `readFile` function now opens a file and extracts all its contents as a lazy character stream:

```
readFile :: FilePath -> IO t String
readFile name =
   openFile name ReadMode (\handle ->
   at handle getAll)

getAll :: Act u File String
getAll =
   eof >>= \b -> if b then return "" else
   getC >>= \c ->
   getAll >>= \s ->
   return (c:s)
```

The operation of `readFile` is depicted in Figure 6.5. Under this scheme, reading the file happens independently from and concurrently to any other operations in the main thread. The function `at` carries out the action `getAll` on the file substate via the reference `handle`. This is regarded as an action on the main state (the real world) in which the rest of the state (the main state `World` value and the other substates) is unaltered. In the main state, the file substate accessible via `handle` is replaced by the final state after the primitive actions `getAll` have been performed. The actions `getAll` are performed lazily, i.e. driven by demand on its result and may be interleaved with later actions in the main thread. For example, in a sequence:

```
main =
  getChar >>= \ch ->
  readFile fname >>= \contents -> putChar ch >>
  putChar ch >>
  putStr contents
```

the file substate is independent of the main state and thus, the order of interleaving does not matter.

Again, since primitive actions are strict in the state, executing `putChar` causes the result state from `readFile` to be evaluated. But, this is the result main state, not the result file substate (as main state actions are not strict in the substates of the main state) and thus, it does not cause operations on

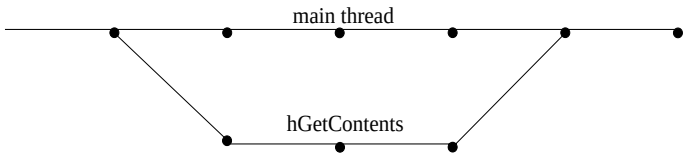



Figure 6.5: Lazy File Read

the substate (i.e `getAll`) to terminate before continuing with any operations in the main state. In `readFile`, the action `at handle getAll` replaces the file substate referred to by `handle` by an unevaluated expression; it is the `putStr contents` or any other action on the file substate that causes `getAll` to be performed. This is all due to the fact that the monadic primitive `thenAct` is lazy.

Also, error checking of the result could cause the `at r f` to terminate. In this case, to guarantee that the actions in the substate are independent of the main action sequence, `f` should normally return success immediately and then do its own error handling.

This contrasts with the approach of Launchbury and Peyton Jones [69] in which unsafe generic interleaving has been used. Here, we rely on the fact that `openFile` is implemented as a safe state-splitting primitive. Consider the following program in standard Haskell:

```
writeFile "file" "x\n" >>
readFile "file" >>= \s ->
writeFile "file" "y\n" >>
putStr s
```

One might expect the final `putStr` to print `x` as the result. However, the contents `s` of the file are read in lazily and are not needed until `putStr` is executed, which is after the second `writeFile`. On all the Haskell systems tested at the time of writing, `putStr` prints `y`, or a file locking error occurs (because the file happens to be still open for reading at the time of the second `writeFile`). This example is not concurrent; there are two state threads, but only one execution thread. The behaviour of the program is unacceptable in a concurrent setting because of the dependency of the behaviour on timings.

Under the state splitting approach, this program does indeed give deterministic results. The second `writeFile` forces the old version of the file to be completely read into memory (or, in a cleverer implementation, moved to a temporary location so that it is not affected by the `writeFile`).

## 6.4 Deterministic Concurrency

### 6.4.1 A Deterministic `fork` Mechanism

The state-splitting proposals we have discussed are independent of the concurrency issue. However, concurrency was the main motivation for studying the problem, so we show here how concurrency is implemented.

A further generic primitive `fork` has been introduced. This is the concurrent counterpart of `at` and has the same type signature:

```
fork ::  Ref t u s1 -> Act u s1 a -> Act t s a
```

The call `fork r f` has the same effect as `at r f`, carrying out an action on a substate of the main state while leaving the remainder of the main state undisturbed. However, `fork` creates a new execution thread to drive the action `f` by evaluating the final substate it produces. This new demand is additional to any other demands on `f` via its result value and the new thread executes independently of and concurrently with, the subsequent actions on the main state, as illustrated in Figure 6.6. Under the deterministic concurrency approach, a processor can run one or more `I/O` performing operations without including any user-visible non-determinism.

### 6.4.2 Lazy File Writing

As an example of the use of `fork`, here is a lazy version of the standard Haskell function `writeFile`. The standard version, unlike `readFile`, is hyperstrict in that it fully evaluates the contents and writes them to the file before continuing with the main state actions. The lazy version shown here evaluates the contents and writes them to the file concurrently with later actions. A separate execution thread is required to drive this, otherwise there might be no demand to cause it to happen.

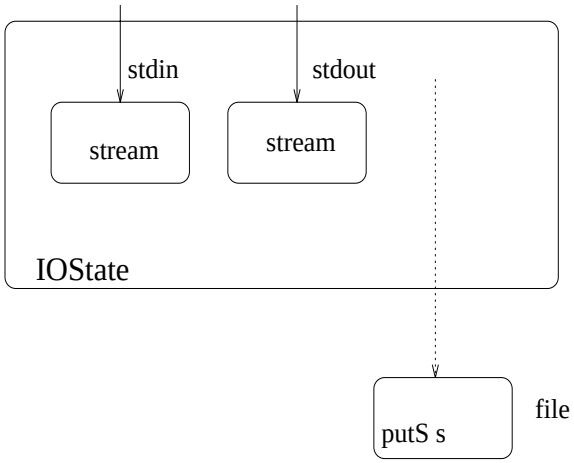



Figure 6.6: State Forking

```
writeFile :: FilePath -> String -> IO t String
writeFile name s =
   openFile name WriteMode >>= \handle ->
   fork handle (putAll s)

putAll :: String -> Act u File ()
putAll s =
   if null s then return () else
   putC (head s) >>
   putAll (tail s)
```

### 6.4.3 Concurrent I/O

In this section, after demonstrating the state splitting/forking ideas, we review the deterministic view of concurrent `I/O`, as it has been adopted by Holyer and Carter [51, 23]. In general, a concurrent program can be viewed as having multiple independent demands from the outside world, each corresponding to a separate action sequence, capable of cooperating with external services. Each external device or service such as a file or window implicitly produces a continuous demand for output. This results in concurrent `I/O` and care needs to be taken in the implementation of concurrency if referential transparency and

determinism are to be preserved.

- Output devices or services are regarded as substates of the outside world and their independence must be ensured. If an additional demand from a new service is created, this splits part of the state and acts on it own substate.

- Different threads should be addressed to independent devices or services, thus non-deterministic effects cannot arise. If two threads go to the same service, say a file manager, which is capable of non-deterministic merge, then the program could produce non-deterministic effects, so it is a designer's obligation to avoid this.

- Aliasing also represents a source of non-determinism and it should be detected. If an action re-opens a file either via the same file or alias, this should disallowed or act as a synchronisation mechanism, forcing all the previous actions to be terminated.

- Each thread should have its own independent non-blocking `I/O`.

- Fairness must also be ensured by time-slicing. Thread switching occurs when a thread's time-slice is over. If one thread is suspended waiting for input, the other threads are indeed able to continue.

- If all threads are blocked waiting for input, this implies the need for a `select` facility in the implementation, for responding to whichever item arrives first. In practice, one can think of a central `I/O` controller which keeps track of all input channels. If a number of threads are waiting for input on separate input channels, the controller responds to the first input to arrive regardless of which item it arrives on. This amounts to a non-deterministic merge. However, all the controller does is to wake up the relevant thread to produce some output. Thus, the program as a whole is not able to produce different results depending on the relative timings at which items arrive on input channels.

In such a setting, the timing of the inputs affects only the timings of the outputs, as with sequential programs. Under such a deterministic concurrency approach,

a processor can run one or more `I/O` performing operations without including any user-visible non-determinism.

On the other hand, the initial proposal for deterministic concurrency assumed that threads do not communicate in the usual style. They only communicate via values they produce as results. However, this model of concurrency cannot cope with current trends in concurrent program designs, such as `GUIs`. Such designs encourage good modularity of components in an object-oriented style, assuming networks of components that communicate via channels. On the other hand, such designs are inherently non-deterministic. To combine such designs with determinism, this thesis expands the initial model of deterministic concurrency, which does not assume any merging mechanism, and investigates deterministic time-merge channels, see Chapter 7.

### 6.4.4 Lazy thenAct and I/O

Of course, using the lazy version of `thenST`, called `thenAct` in the Brisk monadic proposal, required a price to pay. As explained in Chapter 4, in Sections 4.3 and 4.4, it both affects the efficiency of `I/O` and most importantly it disallows infinite `I/O` sequences and its space behaviour needs to be further investigated. But, as already explained in Section 4.4, this problem derives out of a semantic weakness of the monadic `I/O` rather being a problem of the Brisk monadic proposal as such. On the other hand, the use of the lazy version of `thenST` is essential in our setting, e.g. for writing a functional definition of reading the contents of a file lazily (i.e. `hGetContents`), which we considered more important.

## 6.5 Related Work

Kagawa [63] proposed compositional references in order to bridge the gap between monadic arrays [130] and incremental arrays. Compositional references can be also viewed as a means for allowing multiple state types so that a wide range of mutable data structures can be manipulated. On the other hand, they concern only internal facilities and not `I/O`.

The Clean `I/O` system [1] allows a similar mechanism to state splitting, called the multiple environment passing style. This is only possible by using the

Uniqueness Type System of Clean, whereas state splitting does not require any extension to the usual Hindley-Milner type system apart from the second order polymorphism mechanism to ensure safety.

# Chapter 7

# Time-merged Channels for Deterministic Communications

In this chapter, we describe a method which uses hierarchical timestamps on messages to ensure that the merging of messages from different sources into single streams can be carried out in a deterministic way. The key issue is to introduce timing restrictions on the communications, particularly where message streams are merged. The restrictions are not based on real wall-clock time, because maintaining and distributing a universal notion of time would not lead to a satisfactorily complete notion of determinism. Instead, they are based on a hierarchical notion of time which imposes a linear ordering of communication events (messages) within the threads and allows them to be merged in the correct order.

We describe a method which gives each process a status and a time and broadcasts global state information on its output channels in synchronisation messages in a transparent way.

## 7.1 Introduction

When concurrent processes communicate, message streams are merged. This is a major source of non-determinism as merging happens according to accidental times of arrival of messages from different sources, making concurrent systems difficult to reason about and debug.

The problem can be overcome by imposing timing restrictions on the communications, particularly where message streams are merged, so that the merging of messages from different sources into single streams can be carried out in a deterministic way. On the other hand, care needs to be taken so that these restrictions do not reduce concurrency.

In this chapter, we describe a method in which timing information in the form of timestamps is attached to messages. This timing information is used during merging and allows messages to get merged in the correct order. The timestamping is not based on real wall-clock time [67, 46], because maintaining and distributing a universal notion of time is difficult and it would not give a satisfactorily complete notion of determinism. To be suitable also for a distributed setting where no shared global clock is available, timestamping is based on a logical notion of time. This logical notion of time allows messages from different sources to be merged in a deterministic way into a single stream, whereas it affects concurrency to a minimum.

To maximise concurrency, synchronisation messages are used to avoid unnecessary delay in waiting for the possible arrival of earlier messages. This eliminates superfluous busy-waiting and even deadlock.

unnecessary busy waiting or even deadlock caused when delaying messages because earlier messages may yet appear is also avoided through the use of synchronisation messages. If a message arrives at a merge point and it becomes impossible in the system as a whole for any earlier message to arrive at that point, then the message will be forwarded and the process will not be delayed indefinitely.

## 7.2 Hierarchical Time

The idea behind timestamping is that interactions with the outside world are assumed to be represented as a single, linearly ordered stream of incoming messages sent to the processes in the system and may thus be allocated increasing times. Any merging required (e.g. from different devices or users) is assumed to have been carried out already by some other mechanism. Each event triggers a number of further messages within the system to be generated. These are allocated times between the event which triggers them and the following event. The level of a message is just the number of processes which have been involved in triggering it since the original external event.

In order for interpolation of times to be carried out indefinitely, times are described as lists of numbers. Time begins at the start of the program with the empty list, which we will call 0. It is assumed that there is a single ordered stream of messages sent from the outside world into the system. This ordering is reflected in the times of the messages. Thus, the times of the external events are lists of length one which we write 1, 2, 3, .... A process receives a message stamped with time 1 then sends messages stamped with 1.1, 1.2, 1.3 etc (lists of length two) until it next reads in a message. When a further process receives, say, 1.2, the messages which it subsequently sends are stamped with 1.2.1, 1.2.2, 1.2.3 etc. (length three). The ordering is the usual lexicographical ordering on lists. The length of the list of numbers that compose the timestamp of a message defines the number of processes a message has been through. In other words, when a process receives a message with time `t`, it adds an extra component to the time of the message (`clock t`) to indicate that the messages are generated at the next level of time and then it generates messages by incrementing the last component (`clock t`, `tick (clock t)`, `tick (tick (clock t))` ...).

Another way of thinking of these timestamps is that they are like fractional numbers i.e. 1.2.3 is like 0.123, but with arbitrarily sized digits. In this way, all timestamps are put into a linear order which agrees with the usual ordering of real numbers. However, these timestamps differ from decimal numbers in several ways. First, each 'digit' is unlimited in size; time .9 may be followed by arbitrarily many more times which can be represented as .a, .b, .c, and so on. Second, the length of a timestamp is important, so that a process can work out

where to add an extra component digit. Example timestamps are shown here as having digits which count from 1 to avoid any confusion, for human readability, caused by zero digits.

These timestamps may easily be encoded using fixed size digits, e.g. one byte each. Reserve a byte value of 0 for unused bytes. Then, existing code for 0-terminated byte sequences (e.g. C strings) can be used. Alternatively, the bytes may be packed into an array of words, with zeros for the unused bytes in the last word, allowing more efficient word operations to be used without the need to record the exact byte-length of the byte sequence. For compaction, the top bit (as with the UTF-8 encoding of Unicode characters) can be reserved; if the top bit is 1, then the extension mechanism is in force; the number of initial 1s determines the number of bytes used for a component. The bits after the following zero in the first byte and in the subsequent bytes, determine the component number. For example, one byte stores 0 to 127; then a byte starting with bits 10 and the byte following give a 14-bit number where 0 represents 128 and all 1s represents $128 + 2^{14} - 1$ etc. Standard lexical ordering of bytes (or words they are packed into) still works. Also, timestamps may be totally hidden from the applications programmer, implemented in `C` for efficiency. Timestamp comparison is componentwise and standard string comparison can be employed.

```
[] <= us = True
(t:ts) <= [] = False
(t:ts) <= (u:us) = t < u || (t==u && ts <= us)
```

As well as having a linear ordering, times have a tree-structured hierarchy, with more levels of digits added as messages are passed from process to process. This does not prevent cycles from occurring in the pattern of process interconnections. As it is explained, such cycles mean that messages sometimes arrive at a process which have an earlier timestamp than the current time of the process.

## 7.3 Messages

To describe our method, we consider a static network of processes connected via fixed uni-directional channels and communicate via messages, see also Figure 7.1. The network of processes can be described by a directed graph. It is

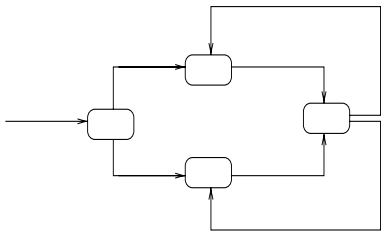


Figure 7.1: Directed Graph of Processes and Channels, With a Merge Point on Entry to Each Process.

assumed also that a single stream of incoming messages is sent to the system from the outside world, where merging from different devices or users has been already carried out by some other mechanism.

In this system, there is no concept of shared memory or any other way for the processes to interact other than via messages. Each message has a timestamp, it gets created by a sender process and is sent only to one process. The same piece of information can be sent to several processes in different messages. Each process knows which processes can send messages to it and merges the messages from these into a single temporally ordered queue before processing them. Also, each process has a time which is always the minimum possible timestamp of the next message which the process expects to send. The messages which a process reads in have timestamps which increase. Communication is asynchronous; when a message is sent, the process that sent it does not suspend.

When a data message e.g. `2` is processed, it may cause a number of descendant messages e.g. `2.1`, `2.2`, `2.3` etc. to be generated in response to it. These children messages are assumed to be generated before the receiver process reads anything else and they are allocated times between the time of the message that triggered then and the time of the following event. When the next message is read in, no more descendent messages from the previous message are generated. This results in an hierarchy of times.

Between sending and receiving messages, processes may carry out internal processing or produce output which goes to the outside world. In keeping with the deterministic concurrency paradigm, the output from different processes is assumed to be addressed to independent devices or services.

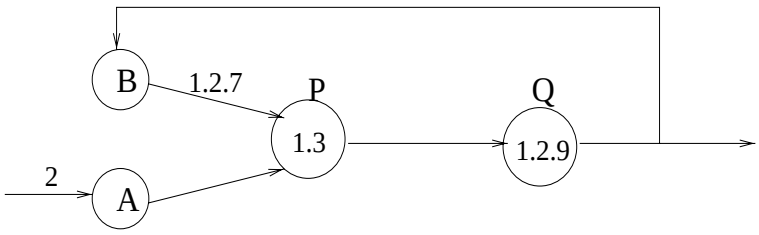



Figure 7.2: Loop

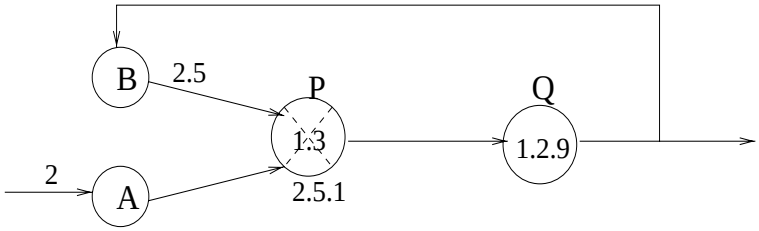



Figure 7.3: Loop

The precise rules of handling timestamps, from the point of view of a single process (ignoring merging), are as follows:

- The messages which a process reads in have timestamps which increase. This is guaranteed by the fact that there is a single stream of incoming messages sent to the system from the outside world.

- The time of a process never decreases. This is the time of the next message which the process expects to send. In other words, when a process sends a message, its current time is attached to the message as a timestamp and then the last digit of the time of the process is incremented. As each process receives always messages with timestamps which increase, guarantees that times increase monotonically.

- Cycles may also occur in the pattern of process interconnections. Such cycles mean that messages sometimes arrive at a process which are earlier than the current time of the process. When a process reads a message with an earlier timestamp than its current time, it does not change its current time. For example, suppose process `P` has two input channels, `A`

and B. It reads in message 1 from channel A, generates responses 1.1 and 1.2 and updates its current time with 1.3. If a message with timestamp 1.2.7 arrives on channel B it gets accepted and any message that comes out will be 1.3.1, see also Figure 7.2. This is explained further below in Section 7.4, Case 2, by assigning each process a status.

- When a process reads a message with a later timestamp than its current time, then its current time is updated to be the same as the timestamp with an extra digit 1 on the end. For example, suppose a process has current time 1.3 and it sends a message. The timestamp on the message is 1.3, and the current time is updated to 1.4. If the process then reads in a message with timestamp 2.5, then the current time of the process is updated to 2.5.1, see also Figure 7.3. This is explained further below in Section 7.4, Case 2, by assigning each process a status.

## 7.4 Merging

A process can have a number of input channels. In this case, receiving a message is equivalent to a non-deterministic select facility. To achieve a deterministic equivalent of a select facility, timestamps are attached to messages which allow a process to choose the earliest message from any of its input channels.

There are two cases to distinguish here: either there are messages on all input channels or not. In the former case, the earliest message can safely be read in and it is guaranteed that this is the one with the earliest timestamp.

When there are no messages in some of the input channels and a process attempts to read in a message, the following problems emerge:

1. **Busy Waiting or Deadlock**

   A process may be suspended if it has not yet been determined whether the first message on the queue has the earliest possible timestamp, or whether an earlier message may yet arrive. Such unnecessary waiting reduces concurrency. Imagine a process P with two input channels, A and B and a message has arrived at a merge point from process A. However, if nothing has arrived at channel B, then the message cannot be safely accepted straight away, because an earlier message may yet arrive on B.

To maximise concurrency, we need to ensure that, once it becomes impossible for an earlier message to arrive on `B`, the process gets to know about it, accepts the message on `A` and continues. To arrange this, each process broadcasts its time on its output channels in synchronisation messages. This allows a process to monitor the time of each of the processes on the other end of its input channels when messages have not been delivered yet. When a synchronisation message arrives at a merge point, it may allow a delayed message to be forwarded.

2. **Recirculation of Earlier Messages**

   Cycles may also occur in the pattern of process interconnections as in Figure 7.2 and cause messages to recirculate and arrive back at the sender process. Such messages have earlier times than the current time of the process. To handle this situation, each process is assigned a status either:

   - *Active*, in which case it is progressing and may send another message in the current sequence at any time;
   - *Waiting* for input, in which case messages earlier than its current time may arrive and thus, the next message it sends may depend on what it receives;
   - *Passive*, in which case no earlier message can arrive and the next expected message will trigger messages with a new sequence of times.

   If a process has status *Active* and time `1.6`, this means that it has sent messages with timestamps `1.1`, ..., `1.5` in response to message `1` and is still processing without reading messages. In this state, it may send a message timestamped `1.6` or may try to read a message and go into status *Waiting* with time `1.6`.

   If a process has status *Waiting* and time `1.6`, this means that it has sent messages `1.1`, ..., `1.5` and is waiting to accept an input message. The message it accepts may have a timestamp $> 1$ and $< 1.6$, i.e. generated indirectly by a message such as `1.2.7` that has been sent earlier and recirculated to arrive back at the same process. In this case, the process may continue the sequence with message `1.6`. Alternatively, it may discover

that all the processes on its input channels have times >=1.6. In this case, no messages generated by 1.5 or earlier can recirculate and arrive back at the process (though there may be such messages elsewhere in the system), so the process goes into *Passive* status with time 2.1. To discover whether no more messages generated by 1.5 or earlier can recirculate and arrive back at the process, global state information needs to be obtained, explained in Point 3 below.

If a process has status *Passive* with time 2.1, this means that no more messages in the sequence 1.n will be generated, 2 is the earliest message which can arrive at the process and the process will respond with a message 2.1 or later.

3. **Cycles of Inactive Processes**

   If there are cycles of inactive (waiting or passive) processes in the process graph, they may cause the recirculating of messages, see also 7.4. A cycle of messages with low timestamps might "get stuck", not recognising that no messages are circulating in it. However, there is also another problem. When a process is in *Waiting* status, it needs to detect when no more messages can recirculate in the current sequence of times in order to increase its time. It is not sufficient for each process to be aware only of the times of its senders and to report the minimum of these as its own time. This raises the need for each process to pass around not only its own time but also full state information, i.e. the times of all its predecessor processes. This state information allows a process to monitor the status of its predecessors and enables the system to tell when a message may be accepted or not, even when messages have not yet appeared in some of its input channels.

   As there is no concept of shared memory in this system, state information can be obtained by sending it along channels within synchronisation messages. Thus, a process needs to broadcast not only its own time on its output channels (point 1) in synchronisation messages but also timing information about its predecessors. Thus, a synchronisation message can be thought of as a set of process identifiers together with their respective

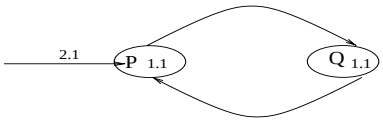



Figure 7.4: Simple Cycle

times.

## 7.5 Algorithm

To describe our method, a static network of processes is considered where processes are connected by uni-directional channels. Each process consists of an identifier, a status, execution code, state, a set of input channels and a set of output channels.

The code of a process may carry on computation, send a message on a given output channel or try to read a message from its input channels.

The state of a process `A` is formed from the process identifiers of the its predecessors, direct or indirect. Each process identifier in the state of process `A` is paired with the earliest time of a message from this process which can arrive at process `A`. This set of pairs (predecessor process identifier, time) is a memory of the last synchronisation message sent; its purpose is only to avoid sending a new synchronisation message if nothing changes. For example, if the state of process `A` is `[("B",t1), ("C",t2)]`, this means that the predecessors of `A` are `B` and `C` and that `t1` and `t2` are the earliest times of messages which can arrive at `A` from `B` and `C` respectively.

For each input channel, the process stores the currently known state of the sender. Whenever in a process, the state of any input channel changes, a new state is recalculated for this process and if it is different from the current state of the process, it is broadcast to its output channels. The recalculation is explained in points 1, 2, 3 below.

We describe an algorithm that allows processes to communicate in a deterministic way by comparing the timestamps on messages during merging, see also Appendix A:

- if the process is *Active* it might:
  - send a message by enqueueing it in the respective output channel after timestamping it with its time. Then the last component of the time of the process is increased by one (e.g. from `1.1` to `1.2`);
  - decide that it will not send any more messages and go in *Waiting* status trying to receive a message;
- if the process is inactive (i.e. has *Passive* or *Waiting* status), then it attempts to receive a message from its input channels. To ensure that merging is deterministic, the process chooses to read from that channel which has the minimum timestamp. To find the channel with the minimum timestamp:
  - first the minimum timestamp in each channel is calculated by checking the currently known state of the sender (stored in the channel).
  - then, the channel with the minimum overall timestamp from all is calculated.

  If the message with the minimum timestamp is a data message, then the state of the sender is set to a synchronisation message, (`Sync [("Sender", t+1)]`) and the process becomes *Active*, otherwise, the status of process remains unchanged.
- for inactive processes, each time an input channel state changes, a process needs to recalculate its state and to broadcast any change of its state. The recalculation consists of the following steps:
  1. first the states of the predecessors of this process need to be merged to form a new state by keeping the minimum timing information for each process;
  2. the new state is then adjusted according to the state of the process. If the status of the process is *Active* the old state of the process is discarded and the new state of the process consists now of the process's own identifier and time. Otherwise, if the new state contains any timing information about the current process, then this entry is

discarded on the basis that the process has become inactive. The timing information about the other processes is then updated by adding one component (e.g. `2.5` becomes `2.5.1`), except that if the current process has *Waiting* status and its time is greater than any of the incoming times, those incoming times are updated to the time of the current process instead and the *Waiting* process becomes *Passive*. For example, if the time of process "P" is 5, the state of its senders is [("P2", 2),("P3", 6),("P1", 1)] and its status is:

- *Passive*, then the adjusted state of the senders becomes [("P2", 2.1), ("P3", 6.1)];
- *Waiting*, then the adjusted state of the senders becomes [("P2", 5), ("P3", 6.1)] and the process becomes *Passive*;
- *Active*, then the adjusted state of the senders becomes [("P1", 5)].

3. if the adjusted new state is different from the state of the process, then the state of the process is discarded, the new adjusted state becomes the state of the process and it is broadcasted to its output channels.

This algorithm ensures that:

1. messages are never accepted out of order;
2. if it is possible for a process to read in a message, it will eventually do so;
3. when no external events arrive for a long time, the system becomes quiescent and does not generate internal synchronisation messages indefinitely.

Point 1 derives from the fact that the time of a process increases always monotonically since a process never sends a message with a timestamp less than its own time. Moreover there are no messages with the same timestamp. The fact that a message is always read from a minimum-time sender ensures that no earlier message can ever arrive.

Point 2 is guaranteed by the fact that when not all input channels of a process have messages, there will be synchronisation messages to inform a process about

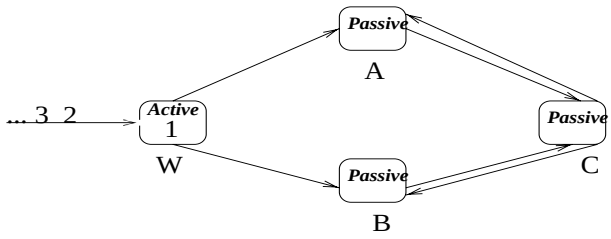



Figure 7.5: A Simple Radio Button

the time of its predecessors and so it can be decided whether a message can be forwarded or not.

Point 3 is guaranteed by the fact that when all predecessor processes have sent information about their time and checked changes against the state of the current process, then the state will not change any more.

## 7.6 A Deterministic Radio Button

Using the above algorithm we show how to implement a deterministic radio button. Imagine a simple radio button consisting of two buttons, `A` and `B`. There is a single streams of incoming messages coming from the outside world `W` and a coordinator `C`, see also Figure 7.5.

```
"W" A 1
"A" P 1
"B" P 1
"C" P 1
"WA" []
"WB" []
"AC" []
"BC" []
"CA" []
"CB" []
```

World sends button press message to button `A` and a synchronisation message to `B`.

```
"W" A 2 ("W",2)
"A" P 1
"B" P 1
"C" P 1
"WA" [M W 1 press]
"WB" [S W 2]
"AC" []
"BC" []
"CA" []
"CB" []
```

Button `A` reads the message and keeps the inferred information about the World. Button `A` sends message to coordinator `C` that it is pressed.

```
"W" A 2 ("W",2)
"A" A 1.2 ("A", 1.2)
"B" P 1 (("W",2.1)
"C" P 1
"WA" []
"WB" []
"AC" [M A 1.1 pressed]
"BC" [S W 2.1]
"CA" []
"CB" []
```

Button `A` goes waiting. Button `C` reads. Button `C` sends message to button `B` to switch off and a synchronisation message to `A`.

```
"W" A 2 ("W",2)
"A" W 1.2 ("W", 2.1)
"B" P 1 ("W",2.1)
"C" A 1.1.2 ("C",1.1.2)
"WA" []
"WB" []
"AC" []
"BC" []
```

```
"CA" [S C 1.1.2]
"CB" [M C 1.1.1 switch_off]
```

Button C goes waiting. Button B also reads and goes waiting. Button A reads and goes passive. Both buttons A and B generate update synchronisation messages.

```
"W" A 2 ("W",2)
"A" P 1.2 ("W", 2.1) ("C",1.2)
"B" W 1.1.1.1 ("W",2.1) ("C",1.1.2.1)
"C" W 1.1.2 ("W",2.1.1)("A",1.2.1)
"WA" []
"WB" []
"AC" [S C 1.2, S W 2.1]
"BC" [S C 1.1.2.1, S W 2.1]
"CA" []
"CB" []
```

Button C reads update synchronisation messages. Button C sends update synchronisation messages to both A and B.

```
"W" A 2 ("W",2)
"A" P 1.2 ("W", 2.1) ("C",1.2)
"B" W 1.1.1.1 ("W",2.1) ("C",1.1.2.1)
"C" W 1.1.2 ("W",2.1.1)("A",1.2.1)
"WA" []
"WB" []
"AC" []
"BC" []
"CA" [S W 2.1.1]
"CB" [S W 2.1.1]
```

Both A and B buttons read synchronisation messages.

```
"W" A 2 ("W",2)
"A" P 1.2 ("W", 2.1)
```

```
"B" P 1.1.1.1 ("W",2.1)
"C" W 1.1.2 ("W",2.1.1)("A",1.2.1)
"WA" []
"WB" []
"AC" [S W 2.1]
"BC" [S W 2.1]
"CA" []
"CB" []
```

The system of processes finally becomes stable and no more synchronisation messages are generated.

```
"W" A 2 ("W",2)
"A" P 1.2 ("W", 2.1)
"B" P 1.1.1.1 ("W",2.1)
"C" P 1.1.2 ("W",2.1.1)
"WA" []
"WB" []
"AC" []
"BC" []
"CA" []
"CB" []
```

## 7.7 Related Work

The most closely related work deals with the problem of maintaining global time in distributed systems.

For example, Birman [11] presented the `ISIS` toolkit, a distributed programming environment based on virtually synchronous process groups and process communication. In `ISIS`, clients communicate with servers either by multicasting or by issuing `RPC` calls. When communicating via multicasting, messages are timestamped according to a notion of vector time, as proposed by Fidge [33] and Mattern [37, 78, 79], where vector time is expressed as a partially ordered system of vectors forming a lattice. This yields a global notion of time: global time is defined as the supremum of all local clock vectors. Although this approach

guarantees consistency, it cannot guarantee determinism, as it is based on an approximation of global time, which is of course, unrepeatable. Furthermore, this timestamping mechanism could not be used to make RPCs deterministic, so avoiding the need for an implicit select mechanism in their implementation [10], as this requires a linear notion of time.

The timestamping mechanism proposed in this chapter is not based on global time and is proposed to work from a more fundamental level, providing an alternative to any implicit select mechanisms used when systems communicate whether via RPC calls or any other conventional communication mechanism.

The timestamping mechanism presented in this chapter timestamps existing communication events, e.g. in a GUI so that an ordering is imposed on them. The additional synchronisation messages mediate timing information when this is not yet available but do not actually play any role in the functionality of the system, so they cannot cause deadlock. In the worse case, if a synchronisation message gets lost, the timing information will not be delivered. Although this might cause an unnecessary delay, it will not lead to a deadlock.

# Part III

# Functional Language Implementation

# Chapter 8

# The Brisk Abstract Machine

This chapter presents the Brisk Machine, a machine model for the implementation of functional languages. It is especially designed to be flexible and dynamic, so that it can support a uniform and efficient implementation of multiple paradigms such as concurrency, computational mobility, dynamic loading and linking and logic programming.

The Brisk Machine is based on the `STG`-machine, though its model is simplified and adapted so that the various paradigms it supports can be accommodated easily without interference between them.

## 8.1 Introduction

### 8.1.1 Why a New Abstract Machine

Many different machine models, described in Section 3.2.2, have been proposed as intermediate forms in the compilation of functional languages, to act as the target of the front end of compilers and as a starting point for code generation. These models represent a great deal of investment in efficiency research with the purpose to provide industrial strength compilers, aiming at making functional languages industrially competitive and generalise the use of functional languages outside of the academia. On the other hand, these performance design decisions make the resulting models complicated and restrict their flexibility and extensibility.

The most prominent machine model that combines the most important features of its predecessors and upon which most of the state of the art Haskell compilers are based, is the `STG-`machine [100]. Although the underlying model of the `STG-`machine is simple, it has been heavily optimised and the assumptions introduced by this tailoring of execution environment make it complicated for adding extra extensions.

Based on the belief that the properties of functional languages allow programs to be well distributed, the challenge has been to provide a tool which could be used in order to extend the functional programming paradigm into different domains such as concurrency and distribution. The Brisk Project[1] [52] has been set up aiming at providing concurrent and distributed working environments where the properties of functional languages, notably referential transparency, are not compromised.

As a consequence of the above mentioned aims, for the Brisk compiler we needed an abstract machine model based on a different philosophy. Since the underlying model of the `STG-`machine is simple, it seemed to be the best starting point. Thus, the Brisk machine is heavily based on the `STG-`machine but has been designed to offer a machine model which is flexible enough to support a number of different run-time execution models. In particular, design considerations have been made in order to support concurrency, distribution and

[1] `Brisk` stands for Bristol Haskell [5] compiler

computational mobility, dynamic loading and linking, debugging tools and logic programming in the form of the Escher [76] extensions to Haskell.

### 8.1.2 Occam's Razor

To achieve a simple machine model and allow extensions to be built without interfering with the optimisations, many issues concerning the execution and also optimisation of execution are relegated to special built-in functions rather than being built into the model itself. This leaves a lean and adaptable basic execution model. As a consequence, the compiler's intermediate language, the *Brisk Kernel Language* (`BKL`), is lower level than other functional intermediate languages. This allows the compiler to perform more low level transformations.[2] A number of extensions and optimisations can be added easily, allowing for the efficient implementation of various computational paradigms.

This simplicity of the `BKL` makes the Brisk machine model extensible and dynamic.

- Extensible, because it allows alternative approaches to evaluation and sharing to be chosen. As a result, extensions concerning optimisations, concurrency, distribution and logic programming have been added without any interference between them.
- Dynamic, because functions are stored as heap nodes. This helps support dynamic loading and computational mobility, where code can either be communicated dynamically between machines or loaded dynamically into the running environment. As a result, this machine model allows for a uniform representation of expressions and functions in the run-time system. A uniform representation of heap nodes, including those which represent functions, allows different execution strategies to coexist, e.g. compiled code, interpretive code and code with logic programming extensions.

[2] It should be mentioned here that `BKL` is not an typed intermediate language. The purpose of a typed intermediate language [104, 137, 97] (Such as `Henk` for Haskell or `Flint` for `ML`, is to offer a general common environment as the single target of the compilation from different source languages, e.g. Haskell, ML. The purpose of `BKL` is to offer an environment where one source language (Haskell) can be compiled into different targets and so distribution or logic programming extensions are supported. `BKL` is a simplified version of the `STG` language used in the Glasgow Haskell compiler.

| | | | |
|---|---|---|---|
| *prog* | $\rightarrow$ | $decl_1$ `;` $\ldots$ `;` $decl_n$ | $n \geq 1$ |
| *decl* | $\rightarrow$ | *fun* $arg_1 \ldots arg_n$ `=` *expr* | $n \geq 0$ |
| *expr* | $\rightarrow$ | `let` $bind_1$ `;` $\ldots$ `;` $bind_n$ `in` *expr* | $n \geq 1$ |
| | $\mid$ | `case` *var* `of` $alt_1$ `;` $\ldots$ `;` $alt_n$ | $n \geq 1$ |
| | $\mid$ | *appl* | |
| *bind* | $\rightarrow$ | *var* `=` *appl* | |
| *alt* | $\rightarrow$ | *con* $arg_1 \ldots arg_n$ `->` *expr* | $n \geq 0$ |
| | $\mid$ | `_ ->` *expr* | |
| *appl* | $\rightarrow$ | *fun* $arg_1 \ldots arg_n$ | $n \geq 0$ |
| *fun* | $\rightarrow$ | *var* | |
| *con* | $\rightarrow$ | *var* | |
| *arg* | $\rightarrow$ | *var* | |

Figure 8.1: The Syntax of the Brisk Kernel Language

## 8.2 The Brisk Kernel Language

The Brisk compiler translates a source program into a minimal functional language called the *Brisk Kernel Language*, or `BKL`. This is similar to functional intermediate languages e.g. the `STG` language [100] used in other Haskell compilers. However, it is lower level than most others, allowing more work to be done in the compiler as source-to-source transformations and thus making the run-time system simpler.

The syntax of `BKL` is shown in Figure 8.1. All mention of types, `data` statements etc. have been omitted for simplicity.

`BKL` provides expressions which are simple versions of *let* constructs, *case* expressions and function applications. Operationally, a *let* construct causes suspension nodes representing subexpressions to be added to the heap, a *case* expression corresponds to a simple switch since its argument is assumed to be evaluated in advance and a function application corresponds to a tail call.

The `BKL` language assumes that evaluation is handled explicitly using primitive functions to be described later. A global function definition represents a single piece of code containing no sub-evaluations, making code generation rel-

atively easy. As a consequence, many complications, including issues to do with evaluation and sharing, are apparently missing from the above picture. These details are hidden away in a collection of built-in functions. This simplifies both `BKL` and the run-time system and allows different approaches to evaluation and sharing to be chosen at different times by replacing these functions. The built-in functions are described more analytically in Sections 8.3 and 8.6.

In order to simplify the run-time system and to allow a uniform representation for expressions, `BKL` has the following features:

1. A *case* expression represents an immediate switch; it does not handle the evaluation of its argument. The argument must represent a node which is guaranteed to be already in head normal form.

2. In `BKL` there is no distinction between different kinds of functions such as defined functions, constructor functions and primitive functions. All functions are assumed here to be represented in a uniform way for simplicity.

3. Every function application must be saturated. That is, every function has a known arity determined from its definition and in every call to it, it is applied to the right number of arguments.

4. In every application, whether in an expression or on the right hand side of a local definition, the function must be in evaluated form. This allows a heap node to be built in which the node representing the function acts as an info node for the application.

5. Local functions are lifted out, so that local *let* definitions define simple variables, not functions. This is described further in 8.3.5.

Evaluation of expressions is not assumed to be implicitly triggered by the use of *case* expressions or primitive functions. Instead, (Points 1, 2) the evaluation of *case* expressions or arguments to strict primitive functions is expressed explicitly using calls to built-in functions such as the `strict` family, discussed below.

The arity restriction (Point 3) means that partial applications are also dealt with explicitly by the compiler during translation into `BKL`, using built-in functions. This frees the run-time system from checking whether the right number

of arguments are provided for each function call and from building partial applications implicitly at run-time.

The saturation of calls and the requirement that functions are in evaluated form before being called (Point 4), allows for a uniform representation of expressions and functions in the run-time system. The motivation is that we want an application `f x y` to be represented by a 3-word heap node in which the first word points to a node representing `f` which also acts as a descriptor for the application node. For this, we need `f` to be an evaluated function node with a special format and not a suspension node. Similarly, every expression can be represented as a node in the form of a function call where the first word points to a function node. Arity information in the function node is always available to describe the call node. Function nodes themselves can all be represented uniformly (Point 2) and in fact follow the same layout as call nodes, being built from constructors as if they were data.

## 8.3 Built-in Functions

The main contribution of the Brisk Machine is that it makes the run-time system match the graph reduction model more closely, by freeing it from many of the usual execution details. In order to achieve this, several issues that arise during execution are handled via a collection of built-in functions. Built-in functions do not only handle constructors and primitive functions, but also issues concerning evaluation and sharing. As a result, the run-time system becomes simple and flexible. Furthermore, several extensions regarding computational mobility as well as logic programming can be treated as built-ins. In the next sections the role of built-in functions in evaluation and sharing is discussed.

### 8.3.1 Arities

In `BKL`, each function has an arity indicating how many arguments its expects. The arity of a function is determined by its definition and in every application of the function, the correct number of arguments must be supplied. The arity does not depend just on the conventional Haskell type of a function, but also on the form of the function's definition. For example, suppose we have the following

Haskell definitions:

```
f x y = x + y
g x = \y -> x + y
```

Assuming that the left hand sides of these definitions are not affected by the compiler during the transformation into BKL, the first will have arity 2 and the second will have arity 1, despite the fact that f and g have the same type. We will indicate this using type declarations which use conventional Haskell syntax, but in which bracketing is taken to be significant. The arity of a function is indicated by the number of unbracketed $\rightarrow$ operators in its type. For example, the functions f and g have types:

```
f :: Int -> Int -> Int
g :: Int -> (Int -> Int)
```

The code generator produces code for a function which matches its arity information. The code for f takes two arguments and returns an integer, whereas the code for g takes one argument and returns a function.

### 8.3.2 Partial Applications

The arity restriction described above means that every function in BKL has an arity indicating how many arguments it expects. This means that every call must be saturated, i.e. the correct number of arguments must always be supplied, according to the arity. This implies that the compiler transformations must introduce explicit partial applications where necessary. For example, given that (+) has arity two, then the expression (+) x has arity one and is translated into the Brisk Kernel Language as a partial application, using the built-in function pap1of2:

```
(+) x  -->  pap1of2 (+) x
```

The function pap1of2 takes a function of arity two and its first argument and returns a function of arity one which expects the second argument. The function pap1of2 is one of a family dealing with each possible number of arguments supplied and expected respectively.

The compiler may also need to deal with over-applications. For example, if `head`, which has arity one, is applied to a list `fs` of functions to extract the first and the result applied to two more arguments `x` and `y`, then the over-application `head fs x y` would need to be translated:

```
head fs x y  -->  let f = head fs in strict100 f x y
```

Since `f` is an unevaluated expression, a member of the `strict` family of functions described below is used to evaluate it before applying it.

This arity restriction also holds for higher order functions. Thus, the compiler also needs to ensure that functions of the right arity are passed to higher order functions. If the function `map` has type:

```
map :: (a->b) -> [a] -> [b]
```

then the compiler determines that the function `map` has arity two and that its first argument has arity one. In this case, the first argument in every call to `map` must be a function with arity one and code is generated for map which assumes this. Thus, `map (+) xs` would have to be transformed:

```
map (+) xs ---> let f x = \y -> x+y in map f xs
```

This transformation can be expressed in `BKL` as:

```
map (+) xs ---> let f x = pap1of2 (+) x in map f xs
```

The compiler has some choice in determining arities when translating a source program into `BKL`. For example, given a definition of the *compose* operator:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
(.) f g = let h x = f (g x) in h
```

the compiler may choose to implement the definition as it stands, giving the operator arity two, or to transform it by adding an extra argument so that it has arity three. It may even be appropriate to compile two versions of the operator for use with different arities.

In the rest of this chapter, the arity information is carried in the type, with brackets to indicate the number of arguments expected. The number of

unbracketed arrows → forms the arity of the function. Thus, if the *compose* operator is compiled with arity two, the type signature above will be used, but if it is compiled with arity three, it would be written as :

```
(.) :: (a -> b) -> (c -> a) -> c -> b
```

The arity restrictions we have described mean that functions are evaluated by calling and returning, as with other values. Other compilers include optimisations which avoid this, implementing a call to a possibly unevaluated function simply as a jump, but the approach we have taken has compensating advantages. It allows for greater uniformity and simplicity in the run-time system; every application can be represented in a direct way, without the need for argument stacks or for testing to see whether enough arguments are available. This argument testing is normally needed to check for partial applications; in Brisk, all partial application issues are dealt with at compile-time.

### 8.3.3 Evaluation and Strictness

The uniform representation of expressions and functions in the run-time system has been one of the main issue in designing the Brisk Machine. This has been achieved meeting two important requirements.

The first requirement is that function applications must be saturated. This has been dealt with by building at compile-time partial applications as explained in the previous paragraph.

The second requirement is that both subexpressions and functions in any function application must be in evaluated form.

- The evaluation of subexpressions is forced by built-in functions introduced by the compiler. Various subexpressions such as the argument to the *case* construct are assumed to be already in evaluated form.

- Moreover, in any function application such as `f x y`, the function `f` is assumed to be in an evaluated form before being called and not to be represented as a suspension node. This enables the expression `f x y` to be represented in the heap as a 3-word node; the first word points to the function node representing `f`. The function node provides layout informa-

tion for the application node, as well as information on how to evaluate it, so `f` must be in evaluated form.

To express the requirement that functions should be in evaluated form more formally, we introduce the notion of *strict type* `!t` for each type `t` to indicate that a variable or a function is in head normal form. This notation has already been used in Haskell for the argument types of strict constructors. For example, in the type signature:

```
f :: !a -> b -> c
```

the function `f` expects two arguments, the first of which has to be in evaluated form as indicated by the `!` symbol. As another example, given:

```
g :: !(a -> b) -> c
```

the function `g` takes as its argument a function which must be in evaluated form.

To force explicit evaluation of expressions, a built-in family of strict functions is provided. For example, `strict01 f x` forces evaluation of `x` before calling `f`. In the family, the 0s and 1s attached to the name `strict` refer to the function and each of its arguments; a 1 indicates that the respective item needs to be evaluated before the function is called and 0 means that no evaluation is required. There is one member of the family for each possible combination of requirements.

The expression `strict01 f x` indicates that in the call `f x`, `f` is assumed to be in evaluated form but `x` has to be evaluated before executing the call `f x`. The type signature of `strict01` is:

```
strict01 :: !(!(!a -> b) -> a -> b)
strict01 f x = ...
```

where `!a -> b` indicates that `f` expects `x` to be in evaluated form, `!(!a -> b)` indicates that `f` must be evaluated before being passed to `strict01` and the outermost `!` indicates that `strict01` itself is assumed to be in evaluated form. An example which illustrates the use of the strict family of functions is the definition of `(+)`:

```
(+) :: !(Int -> Int -> Int)
x + y = strict011 (!+) x y

(!+) :: !(!Int -> !Int -> Int)
x !+ y = ...
```

Here `strict011` indicates that both `x` and `y` need to be evaluated before making the call `(!+) x y`. The `(!+)` function is a built-in one which assumes that its arguments are already evaluated.

To explain the operation of the `strict` functions, take `strict011 (!+) x y` as an example. Operationally `strict011` places a pointer to the original expression `strict011 (!+) x y` on a return stack with an indication that `x` is under evaluation (see Section 8.5.2) and makes `x` the current node. When `x` is evaluated to `x'` (say), the expression is updated in place and becomes `strict001 (!+) x' y` with an indication that `y` is under evaluation and `y` is made the current node. When `y` is evaluated to `y'`, the expression is popped off the stack and updated to `(!+) x' y'` which becomes the current node. The operation of `strict` functions is further described in Section 8.6.

An alternative approach which allows a little more optimisation is to treat the `strict` functions as having arity one. Then `(+)` would be then defined by `(+) = strict011 (!+)` and the compiler has the opportunity to generate code for `(+)` directly, inlining the call to `strict011`.

One benefit of using the `strict` family of functions to describe evaluation and of using built-in functions generally, is that they can be replaced by alternative versions which represent different evaluation strategies (e.g. for mobility issues or in logic programming extensions to the language).

Furthermore, if the notation `!t` were to be made available to programmers, there would need to be restrictions placed on it so that the type-checker could prevent any misuse. The main restrictions would be that polymorphic type variables should not be allowed to range over strict types, and that results of functions should not have strict types. These two restrictions ensure that suspensions do not have strict types. For example, consider the expression:

```
let x = id y in ...
```

Since `x` is represented as a suspension, it should not have a strict type. The identity function `id` has polymorphic type `a -> a`, so if `y` were allowed to have a strict type, then `x` could be given a strict type, which is prevented from the restrictions.

The notation for strict types that we have introduced here can be extended to deal with unboxing [103], by associating a suitable representation with each strict type. The issue of unboxing in the compiler and the Brisk Kernel Language are not discussed further here. However, the run-time system and abstract machine described later allow for unboxed representations.

### 8.3.4 Sharing

One way to implement sharing is to introduce the built-in function `share` which ensures that a subexpression is evaluated only once. The call `share x` is equivalent to `x`, but when it is first accessed an update frame is put on the return stack while `x` is evaluated. When the evaluation of `x` finishes, the update frame overwrites the call `share x` with an indirection to the result. An example which illustrates this approach to sharing is the `map` function with definition:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Translating this into the Brisk Kernel Language gives:

```
map f xs = strict001 map' f xs


map' f xs = case xs of
   [] -> []
   (:) y ys ->
      let z = share z'
          z' = strict10 f y
          zs = share zs'
          zs' = map f ys
      in (:) z zs
```

The definition of `map` uses a member of the `strict` family to specify that the second argument to `map` needs to be evaluated before pattern matching can

occur. Also in building a suspension for `f y`, the function `f` is not known to be evaluated and laziness dictates that it should not be evaluated before it is called, so the suspension `strict10 f y` is created.

In `map'`, the variable `z` is defined as a shared version of `z'` and `z'` is defined as `strict10 f y` because `f` is a dynamically passed function, which may not be in evaluated form. The variable `z'` is not mentioned anywhere else, so that the reference to it from `z` is the only reference. All other references point to the "share node" `z`. When `z` is evaluated, the built-in `share` function arranges for `z'` to be evaluated and for the share node to be updated to become an indirection to the result. Thus all references now share the update.

This approach to sharing has a high space overhead. In the absence of sharing analysis, every node representing an unevaluated expression, i.e. every application of a function which is not a constructor, has to be assumed to need sharing and for every such node a 2-word share node is added. An alternative optimised approach to sharing is to build it into the return mechanism, where the original expression is kept on the stack instead of keeping one copy of it. Every time the current node holding an argument of this expression becomes evaluated, the return mechanism uses the top entry on the stack to check whether the result is different from the original expression node. The checking involves to find out which node caused evaluation of the current subexpression and which word in that node points to the original version of the subexpression. If the result is different from the original node, it overwrites the original expression node with an indirection. When all arguments are evaluated, the function in the original call, which is kept on the stack, is overwritten with its continuation and popped off the stack. The overhead of this test on every return is offset by the fact that far fewer update frames need to be built – they are only needed to cover certain special cases.

Updating has also been added to the return mechanism by over-writing the original version of the subexpression with an indirection to the result, after a quick comparison test to ensure that the result node is actually different from the original one. The overhead of this test is compensated for by the fact that no time or space is wasted in constructing update frames.

### 8.3.5 Lifting

Lifting is an important issue in the Brisk compiler, since there are no local function definitions in BKL; local definitions represent values not functions. In Brisk, it is possible to support either the traditional approach to lifting, or the approach taken in the Glasgow compiler; we describe both briefly here.

In the traditional approach to lifting [58], the free variables (or maximal free subexpressions) of a local function are added as extra arguments, so that the function can be lifted to the top level. The local function is then replaced by a partial application of the lifted version. For example, given a local definition of a function f:

```
... let f x y = ... g ... h ... in e
```

two extra arguments g and h are added and the definition is lifted out to the top level as f':

```
... let f = pap2of4 f' g h in e


f' g h x y = ... g ... h ...
```

An explicit partial application is built using pap2of4 in order to preserve the arity restrictions of BKL.

Peyton Jones & Lester [72] describe a slightly different approach to lifting. For a local function such as f, a node is built locally to represent f and this node contains references to the free variables of f. The code for f can be lifted out to the top level; it obtains the free variables from the node representing f and obtains the other arguments in the normal way.

To express this form of lifting in BKL, the language can be extended so that free variables are represented explicitly using a tuple-like notation. For example, the lifted version of f above becomes:

```
... let f =  f' (g, h) in e


f' (g,h) x y = ... g ... h ...
```

The local definition of f represents the idea that f is built directly as a function node, using f' as its code and including references to g and h. The definition

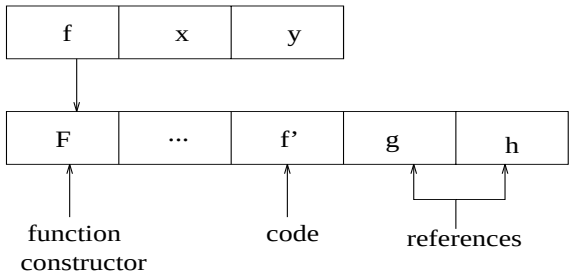



Figure 8.2: A Function with References and Arguments

of `f'` indicates that self-contained code should be generated which gets `g` and `h` from the function node and `x` and `y` from the call node, as shown in Figure 8.2. This approach to lifting fits in well with the way in which global functions are treated in Brisk. In order to support dynamic movement of code, a global function is represented as a node in the heap containing explicit references to the other global values which it uses. Thus a global definition:

```
f x y = ... g ... h ...
```

can also be represented in the extended notation as:

```
f = f' (g,h)

f' (g,h) x y = ... g ... h ...
```

indicating that `f` is represented as a node referencing `g` and `h` and that self-contained code is generated for `f'`.

## 8.4 The Brisk Abstract Machine

Many approaches to compiling functional languages [59], [6], [136], [15], [105], use abstract machines with imperative machine instructions to provide the operational semantics of programs; see also [14], [99]. With the `STG` Machine [100], a different approach was taken. The `STG` language is a small, low level, intermediate language which is on the one hand still functional and yet on the other can be given a direct operational state transition semantics to guide its translation into `C` or machine code.

The Brisk compiler uses the intermediate language BKL which, like STG, is a low level functional language. A direct semantics could similarly be given for it. However, the computational mobility issues which Brisk addresses mean that BKL needs to be compiled into interpretive bytecode in some circumstances and C or machine code in others.

We have chosen to present the operational semantics of BKL constructs using a conventional abstract machine similar to the original G-machine, so that the machine instructions correspond very closely to the bytecode instructions produced via the interpretive compiling route. They also guide direct translation into C or machine code and provide a link with the design of the run-time system.

The abstract machine described here is called the *Brisk Abstract Machine*, or BAM for short. First, we show how to compile the Brisk Kernel Language into BAM code and then we give the semantics of the individual instructions of the BAM code in terms of the run-time system described above.

The Operational semantics of the built-in functions are also given in Section 8.6 using conventional state transition systems.

### 8.4.1 Compiling into BAM Code

The state of the Brisk Abstract Machine consists of a *frame* of local variables in addition to the return stack and the heap. The frame contains the local variables used during execution. It is completely local to each individual function call and starts out empty. It is extended in a stack-like manner as needed. This stack-like use is very similar to the stack in the original G-machine and it allows the BAM instructions to have at most one argument each, often a small integer, which leads to a compact bytecode.

On the other hand, the BAM frame is not used as an argument stack or a return stack, items don't have to be popped off so frequently and it does not have to be left empty at the end of a function call. This allows us to use it as an array as well a stack, indexed from its base. Initial instructions in the code for each function indicate the maximum size of frame needed and the maximum amount of heap space to be used, but we omit those here. The function $\mathcal{F}$ compiles code for a function definition:

$$
\begin{array}{l}
\mathcal{F}\ [\![f\ (g_1,\ldots,g_n)\ x_1 \ldots x_m = e]\!] = \\
\quad [\texttt{GetRefs}\ n, \texttt{GetArgs}\ m]\ ++ \\
\quad \mathcal{C}\ [\![e]\!]\ [g_1 \mapsto 0, \ldots, g_n \mapsto n-1, x_1 \mapsto n, \ldots, x_m \mapsto n+m-1]\ (n+m) \\
\qquad ++\ [\texttt{Enter}]
\end{array}
$$

A function $f$ with references to free variables $g_i$ and arguments $x_i$ is compiled into code which first loads the references and arguments into the frame, then compiles code for the right hand side $e$ which creates a pointer to a node representing $e$ and then ends by entering the code for this result node.

The function $\mathcal{C}$ generates code for an expression. A call $\mathcal{C}\ [\![e]\!]\ \rho\ n$ generates code for $e$, where $\rho$ is an environment which specifies where the variables appear in the frame and $n$ is the current size of the frame.

The code generated for a variable or a constant is:

$$
\begin{array}{l}
\mathcal{C}\ [\![x]\!]\ \rho\ n = [\texttt{GetLocal}\ (\rho\, x)] \\
\mathcal{C}\ [\![c]\!]\ \rho\ n = [\texttt{GetConst}\ c]
\end{array}
$$

The `GetLocal` instruction gets a variable from its position in the frame and pushes it onto the end of the frame. The `GetConst` instruction pushes a one-word unboxed value onto the end of the frame. The frame is thus untyped; it contains both pointers and raw values. This does not cause any problems since the frame only lasts for a single function call so, for example, the garbage collector need not be concerned with it.

The code generated for an application is:

$$
\begin{array}{l}
\mathcal{C}\ [\![f\ x_1 \ldots x_n]\!]\ \rho\ m = \\
\qquad \mathcal{C}\ [\![x_n]\!]\ \rho\ m\ ++ \ldots ++\ \mathcal{C}\ [\![x_1]\!]\ \rho\ m\ ++ \\
\qquad \mathcal{C}\ [\![f]\!]\ \rho\ m\ ++ \\
\qquad [\texttt{Mkap}\ (n+1)]
\end{array}
$$

The code just gathers together the function and its arguments and then creates a new application node in the heap from them.

The code generated for a `let` construct is:

$$
\begin{array}{l}
\mathcal{C}\ [\![\texttt{let}\ y_1 = ap_1; \ldots; y_n = ap_n\ \texttt{in}\ e]\!]\ \rho\ m = \\
\quad [\texttt{Alloc}\ k_1, \ldots, \texttt{Alloc}\ k_n] \\
\quad \mathcal{D}\ [\![y_1 = ap_1]\!]\ \rho'\ m' +\!\!+ \ldots +\!\!+\ \mathcal{D}\ [\![y_n = ap_n]\!]\ \rho'\ m' +\!\!+ \\
\quad \mathcal{C}\ [\![e]\!]\ \rho'\ m' \\
\quad \text{where} \\
\qquad k_i \text{ is the size of application } ap_i \\
\qquad \rho' = \rho\ [y_1 \mapsto m, \ldots, y_n \mapsto m + n - 1] \\
\qquad m' = m + n
\end{array}
$$

The code generated here assumes that the `let` construct is recursive. As usual, if the `let` construct is known to be non-recursive, simpler code can be generated, but we don't show that here. First, space is allocated on the heap for each node to be created and the variables $y_i$ are made to point to these blank nodes. Then each node is filled in in turn using the $\mathcal{D}$ function:

$$
\begin{array}{l}
\mathcal{D}\ [\![y = f\ x_1 \ldots x_n]\!]\ \rho\ m = \\
\quad \mathcal{C}\ [\![x_n]\!]\ \rho\ m +\!\!+ \ldots +\!\!+\ \mathcal{C}\ [\![x_1]\!]\ \rho\ m +\!\!+ \\
\quad \mathcal{C}\ [\![f]\!]\ \rho\ m +\!\!+\ \mathcal{C}\ [\![y]\!]\ \rho\ m +\!\!+ \\
\quad [\texttt{Fill}\ (n+1)]
\end{array}
$$

This generates code to fill in a blank node by gathering the function and its arguments, together with the pointer to the node to be filled and then using the `Fill` instruction to fill in the node.

The code generated for a `case` construct is:

$$
\begin{array}{l}
\mathcal{C}\ [\![\texttt{case}\ v\ \texttt{of}\ alt_1 \ldots alt_n]\!]\ \rho\ m = \\
\quad [\texttt{Table}\ k]\ +\!\!+ \\
\quad \mathcal{A}\ [alt_1]\ \rho\ m\ +\!\!+ \ldots +\!\!+\ \mathcal{A}\ [alt_n]\ \rho\ m\ +\!\!+ \\
\quad [\texttt{Switch}\ (\rho\, v)] \\
\quad \text{where } k \text{ is the number of constructors in the type of } v
\end{array}
$$

The code first allocates a jump table with one entry for every constructor in the type of $v$. This is followed by code for each alternative, generated using $\mathcal{A}$. An interpreter for the code scans the alternatives, filling in the jump table. Once the jump table is complete, the `case` variable is used by the `Switch` instruction to jump to the appropriate case.

The code generated for an alternative is:

$\mathcal{A}\ [\![C\ x_1 \ldots x_n \texttt{->}\ body]\!]\ \rho\ m =$

$\quad [\texttt{Case}\ i, \texttt{Skip}\ b, \texttt{Split}\ n]\ ++$

$\quad \mathcal{C}\ [\![body]\!]\ \rho'\ m'\ ++$

$\quad [\texttt{Enter}]$

$\quad$ where

$\qquad i$ is the sequence number of constructor $C$

$\qquad b$ is the amount of code generated for the body

$\qquad \rho' = \rho\ [x_1 \mapsto m, \ldots, x_n \mapsto m + n - 1]$

$\qquad m' = m + n$

$\mathcal{A}\ [\![\_ \texttt{->}\ body]\!]\ \rho\ m =$

$\quad [\texttt{Default}, \texttt{Skip}\ b]\ ++$

$\quad \mathcal{C}\ [\![body]\!]\ \rho\ m\ ++$

$\quad [\texttt{Enter}]$

The `Case` instruction is used to fill in an entry in the previously allocated jump table. The entry points to the code for the body of the alternative, just after the `Skip` instruction. The `Skip` instruction skips past the code for the alternative (including the `Split` and `Enter` instructions). The `Split` instruction loads the constructor arguments into the frame and the environment is extended with their names. A default `case` alternative causes all the unfilled entries in the jump table to be filled in.

### 8.4.2 The BAM Instructions

Here, we give transition rules which describe the action of each of the instructions on the run-time system state. The state consists of the current sequence of instructions $i$, the frame $f$, the return stack $s$ and the heap $h$. When it is needed, the current node will be indicated in the heap by the name $cn$.

The frame will be represented in a stack-like way as $f = x : y : z : \ldots$, but indexing of the form $f!i$ will also be used, with the index being relative to the base of the stack.

The `GetLocal` $n$ instruction pushes a copy of the n'th frame item onto the end of the frame:

| | | | | | |
|---|---|---|---|---|---|
| `GetLocal` $n : i$ | | $f$ | $s$ | $h$ |
| | $i$ | $f!n : f$ | $s$ | $h$ |

The `GetConst` $c$ instruction pushes a constant onto the end of the frame:

| | | | | |
|---|---|---|---|---|
| `GetConst` $c : i$ | | $f$ | $s$ | $h$ |
| | $i$ | $c : f$ | $s$ | $h$ |

The `GetRefs` $n$ instruction loads $n$ references into the frame, from the function node mentioned in the current node:

| | | | | | |
|---|---|---|---|---|---|
| `GetRefs` $n : i$ | | $f$ | $s$ | $h$ | $\left[\begin{array}{l} cn \mapsto < g, \ldots > \\ g \mapsto < \ldots, g_1, \ldots, g_n > \end{array}\right]$ |
| | $i$ $g_n : \ldots : g_1 : f$ | | $s$ | $h$ | |

The `GetArgs` $n$ instruction extracts $n$ arguments from the current node into the frame:

| | | | | |
|---|---|---|---|---|
| `GetArgs` $n : i$ | | $f$ | $s$ | $h[cn \mapsto < g, x_1, \ldots, x_n >]$ |
| | $i$ $x_n : \ldots : x_1 : f$ | | $s$ | $h$ |

The `Mkap` $n$ instruction assumes that the top of the frame contains a function and its $n-1$ arguments, from which a node of size $n$ is to be built:

| | | | | |
|---|---|---|---|---|
| `Mkap` $n : i$ | $g : x_1 : \ldots : x_{n-1} : f$ | | $s$ | $h$ |
| $i$ | | $p : f$ | $s$ | $h[p \mapsto < g, x_1, \ldots, x_{n-1} >]$ |

The `Alloc` $n$ instruction allocates space for a node of size $n$ on the heap, with uninitialised contents:

| | | | |
|---|---|---|---|
| `Alloc` $n : i$ | $f$ | $s$ | $h$ |
| $i$ | $p : f$ | $s$ | $h[p \mapsto <?_1, \ldots, ?_n >]$ |

The `Fill` $n$ instruction fills in a previously allocated node, given its pointer and contents:

| | | | | |
|---|---|---|---|---|
| `Fill` $n : i$ | $p : g : x_1 : \ldots : x_{n-1} : f$ | | $s$ | $h[p \mapsto <?_1, \ldots, ?_n >]$ |
| $i$ | | $f$ | $s$ | $h[p \mapsto < g, x_1, \ldots, x_{n-1} >]$ |

To implement the `case` construct, a jump table is needed. We temporarily extend the state, adding the table as an extra component $t$. The `Table` $n$ instruction allocates an uninitialised table of size $n$:

| | | | | | |
|---|---|---|---|---|---|
| `Table` $n : i$ | $f$ | $s$ | $h$ | | |
| $i$ | $f$ | $s$ | $h$ | $t \mapsto <?_1, \ldots, ?_n>$ | |

Each `Case` $k$ instruction causes the $k$'th table entry to be filled in with the position $i$ in the instruction sequence, just after the subsequent `Skip` instruction:

| | | | | | | |
|---|---|---|---|---|---|---|
| `Case` $k$ : `Skip` $b : i$ | $f$ | $s$ | $h$ | $t \mapsto <\ldots, ?_k, \ldots>$ | |
| `Skip` $b : i$ | $f$ | $s$ | $h$ | $t \mapsto <\ldots, i, \ldots>$ | |

A `Default` instruction causes any unfilled entries in the table to be filled in with the relevant position in the instruction sequence:

| | | | | | | |
|---|---|---|---|---|---|---|
| `Default` : `Skip` $b : i$ | $f$ | $s$ | $h$ | $t \mapsto <\ldots, ?, \ldots, ?, \ldots>$ | |
| `Skip` $b : i$ | $f$ | $s$ | $h$ | $t \mapsto <\ldots, i, \ldots, i, \ldots>$ | |

Once the table has been filled in, the `Switch` instruction gets the `case` variable $v$ from the frame and causes a jump to the relevant alternative. The sequence number $k$ of the constructor $C$ used to build $v$ is used to index the jump table. The sequence number of $C$ can be recorded in its heap node. Execution continues with position $i_k$ in the instruction sequence, where $i_k$ is the $k$'th table entry. The table is not needed any more:

| | | | | | | |
|---|---|---|---|---|---|---|
| `Switch` $n$ : $i$ | $f$ | $s$ | $h[f!n \mapsto <C, \ldots>]$ | $t \mapsto <\ldots, i_k, \ldots>$ | |
| $i_k$ | $f$ | $s$ | $h$ | | |

While the jump table is being built, the `Skip` instruction is used to skip past the code for the alternatives. The instruction `Skip` $b$ simply skips over $b$ instructions (or $b$ bytes if the instructions are converted into bytecode):

| | | | | |
|---|---|---|---|---|
| `Skip` $b$ : $i$ | $f$ | $s$ | $h$ | |
| $drop\ b\ i$ | $f$ | $s$ | $h$ | |

At the beginning of the code for a `case` alternative, the instruction `Split` $n$ is used to extract the fields from the `case` expression, which is still at the top of the frame at this moment:

$$\begin{array}{llllll} \texttt{Split}\ n : i & & v : f & s & h[v \mapsto < C, x_1, \ldots, x_n >] \\ \quad i & x_n : \ldots : x_1 : f & & s & h \end{array}$$

At the end of the code for a function, or at the end of an alternative, the `Enter` instruction transfers control to a new function. The node on the top of the frame becomes the current node, a new instruction sequence is extracted from its function node and execution of the new code begins with an empty frame:

$$\begin{array}{llllll} \texttt{Enter} : i & p : f & s & h \left[ \begin{array}{l} p \mapsto < g, \ldots > \\ g \mapsto < \ldots, fi, \ldots > \end{array} \right] \\ \quad fi & [] & s & h\ [cn = p] \end{array}$$

Instructions that manipulate the return stack are handled by special functions such as constructors and the *strict* family, rather than being produced by the compiler. As an example, the code for a constructor is [`Return`, `Enter`] where the `Return` instruction puts the current node into the appropriate argument position of the previous node and pops the return stack:

$$\begin{array}{llllll} \texttt{Return} : i & [] & < k, p > : s & h \left[ \begin{array}{l} p \mapsto < \ldots, x_k, \ldots > \\ cn \mapsto < \ldots > \end{array} \right] \\ \quad i & [p] & s & h\ [p \mapsto < \ldots, cn, \ldots >] \end{array}$$

The operations of the *strict* family of functions is described in detail in paragraphs 8.5.2 and 8.6.

## 8.5 The Brisk Run-Time System

In order to complement the design of the Brisk machine, in this section we describe the way in which programs are represented in Brisk. More details about the Brisk run-time system are given in chapter 9. The representation is based heavily on the one used for the `STG` Machine. However, it is simplified and made more uniform so that the link with conventional simple graph reduction is as clear as possible, making it easier to adapt the run-time system to alternative uses. At the same time, enough flexibility is retained to allow most of the usual optimisations to be included in some form.

The state of a program consists of a heap to hold the graph, a current node pointer to represent the subexpression which is currently being evaluated and a return stack which describes the path from the root node down to the current node. There are no argument or update stacks.

### 8.5.1 The Heap

The heap contains a collection of nodes which holds the program graph. Every node in the heap can be thought of as a function call. It consists of a function pointer followed by a number of raw words, followed by a number of pointer words. A spineless representation is used; nodes vary in size, with the number of arguments being determined by the arity of the function. For example, an expression `f x y z` is represented as a 4-word node in which the first word points to a node representing `f` and the other words point to nodes representing `x`, `y` and `z`.

Another characteristic of the Brisk run-time system is that there is a single *root node* representing the current state of the program as a whole and all active nodes are accessible from it. Free space is obtained at the end of the heap and a copying or compacting garbage collector is used to reclaim dead nodes.

In the Brisk Machine, functions are represented the same way as call nodes. A node representing a function contains information about the arity of the function, the code for evaluating the function, code for other purposes such as garbage collection or debugging and references to other nodes needed by the evaluation code, i.e. the free variables. The arity information allows the function node to be used as an *info node*, i.e. one which describes the size and layout of call nodes. To ensure that the size and layout information is always available, the function pointer in a call node must always refer to an evaluated function node and not to a suspension which may later evaluate to a function.

In general, functions may have both unboxed and boxed arguments, with the unboxed ones preceding the boxed ones and the arity of a function reflects how many of each. A node may thus in general consist of a function pointer followed by a number of raw words, followed by a number of pointer words. Constructors with unboxed arguments can be used to represent raw data. A large contiguous data structure such as array can be stored in a single node of

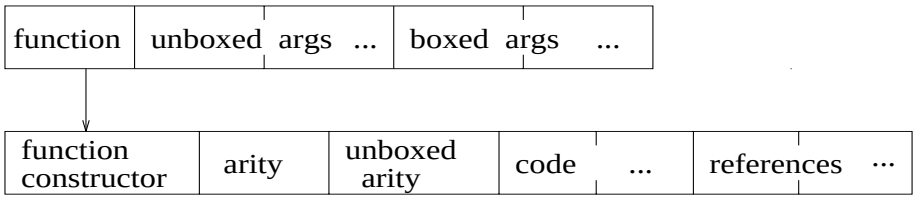



Figure 8.3: A Node as a Function Call

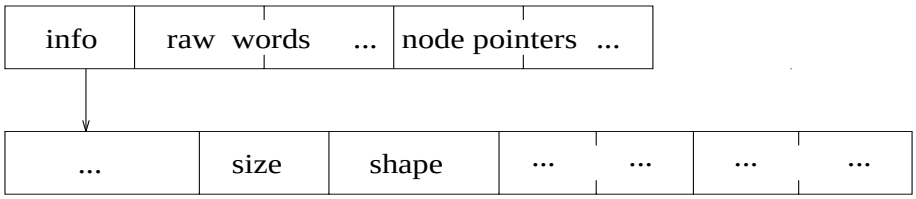



Figure 8.4: A Node as a Uniform Data Structure

any size; a suitable constructor node can be generated dynamically to act as the function pointer.

This uniform representation of nodes means that any heap node can be viewed in three ways. First, a node represents an expression in the form of a function call, as in Figure 8.3:

Second, a node can be treated as a data structure which can be manipulated, e.g. by the garbage collector or by debugging tools, in a uniform way, as in Figure 8.4:

Third, a node can be regarded as an active object, responsible for its own execution, with methods for evaluation and other purposes available via its info pointer, as in Figure 8.5.

Function nodes follow the same pattern. The function pointer at the be-

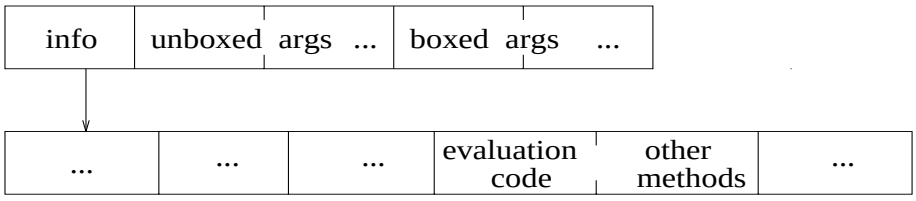



Figure 8.5: A Node as an Object

ginning of a function node can be regarded as a constructor, with functions being thought of as represented using a normal data type, hidden from the programmer by an abstraction. Such a function constructor has a number of unboxed arguments representing arity and code information and a number of boxed arguments representing references to other nodes. All functions, including global ones, are represented as heap nodes which contain references to each other. New info nodes can be created dynamically to cope with special situations, newly compiled code can be dynamically loaded and code can be passed from one process to another, which provides greater flexibility. This contrasts with other systems [100], [25], where info structures are stored statically, which would prevent such dynamic movement of nodes.

It is becoming common, e.g. in Java [41], to use an architecture independent bytecode representation of program code, which makes it portable. This can be used to ship programs over a network and interpret them remotely. The same technique is used in Brisk. However, statically compiled code is also supported. To allow both types of code to be mixed in the same heap, the code word in a function node points to a static entry point. In the case of statically compiled code, this is precisely the evaluation code for the function. In the case of bytecode, the code word points to the static entry point of the interpretive code and the function node contains an extra reference to a data node containing the bytecode.

### 8.5.2 The Return Stack

The *return stack* in the Brisk run-time system keeps track of the current evaluation point. The basic idea for the return stack is that each stack entry consists of a pointer to a node in the heap representing a suspended call and the position within that node of an argument which needs to be evaluated before execution of the call can continue, as in Figure 8.6.

In the compiler, each stack node (frame) consists of a function constructor and four arguments. The fourth argument points to the original node in the heap representing a suspended call, in which one of the fields is being evaluated. The first argument is an unboxed argument which specifies the position within that node of an argument which needs to be evaluated before execution of the

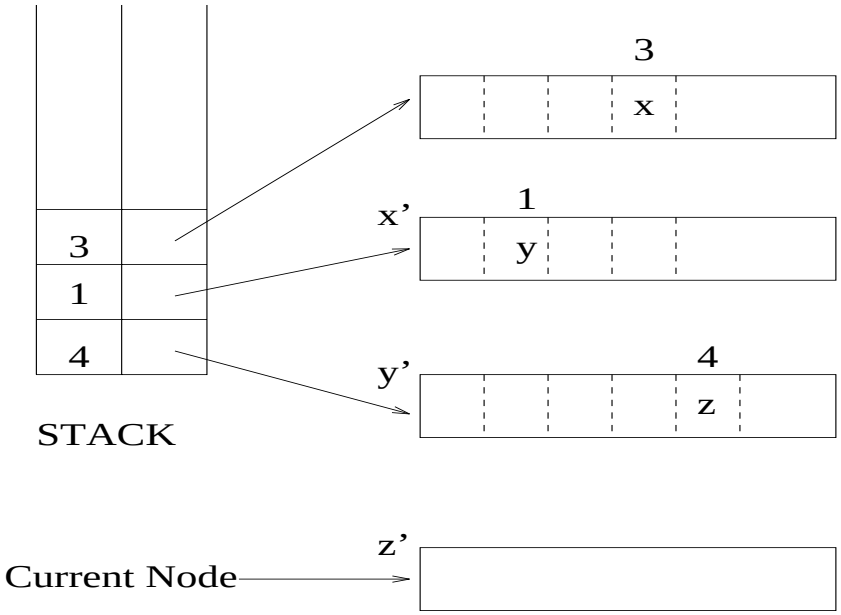



Figure 8.6: The Return Stack

call can continue. The second argument chains the stack frames together; it points to the next frame. The third argument is a function that acts as a continuation function. This continuation function will replace the one in the original node, once evaluation of all the fields in the pattern is complete. This is depicted and explained further in figures of Section 8.6.

An evaluation step consists of calling the evaluation code for the current node. If the current node is already in head evaluated form, the code causes an immediate return. Otherwise, the code carries out some processing, usually ending with a tail call. Notionally, a tail call consists of building a new node in the heap and making it current.

When the evaluation of the current node is complete, i.e. the current node is in head normal form, a return is performed. This involves plugging the current node into the previous node (pointed to by the top of the stack) at the specified argument position, making the previous node into the new current node and popping the stack.

However, tail calls lead to a large turnover of heap space, with nodes being created for every intermediate expression. As an optimisation to reduce the

number of heap nodes created, a tail call can create a temporary node; this is a node allocated in the usual way at the end of the heap, but without advancing the free space pointer. This node will then usually immediately overwrite itself as it in turn creates new heap nodes. To support this, certain nodes such as data nodes (constructors) or functions which save pointers to the current node in stack frames or elsewhere, or functions which reuse the current node as a space optimisation must make themselves permanent where necessary by advancing the free space pointer before continuing execution. This simple scheme provides many of the same advantages as more complex systems which use registers and stacks for temporary storage of intermediate expressions.

## 8.6 The Operational Semantics of the Built-ins

Built-in functions play a crucial role in the functionality of the Brisk Machine, as they simplify the run-time system and allow extensions and optimisations to be built naturally without any interference between them. In the previous paragraphs, the operational semantics of the built-ins have been already explained informally. In this section, a formal operational semantics for built-ins that deal with evaluation and partial application is given. Unlike the Brisk Kernel Language constructs, the operational semantics of which have been expressed using an abstract machine, the operational semantics of the built-ins are expressed using the state of a program which consists of:

- a heap to hold the graph;
- a current node pointer `cn` to represent the subexpression which is currently being evaluated;
- a return stack which describes the path from the root node down to the current node.

### 8.6.1 Evaluation

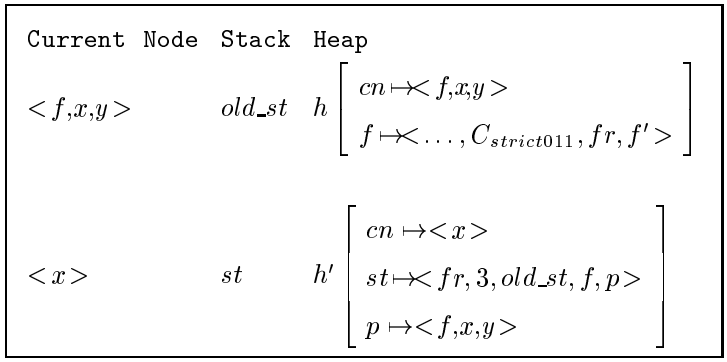
As already explained, evaluation is forced by the built-in family of strict functions. A call `f x y` is represented in the heap as a 3-word node `<f,x,y>` in

which the first word points to a node representing `f` which acts as an info node for the call node. This info node has the form:

f $\mapsto <\ldots, C_{strict011}$,fr,f'>

Here $C_{strict011}$ is the evaluation code and `fr` and `f'` are the references of the call node `<f,x,y>`. Reference `fr` is a stack frame constructor and reference `f'` is a continuation function that will replace the one in the original node, once evaluation of all the fields in the pattern is complete. Again the member `strict011` of the strict family of functions implies a function `f` in evaluated form of two arguments, `x` and `y`, both of which require evaluation.

The stack frame constructor `fr` creates stack nodes. An example of a stack node is `<fr,3,old_st,f,p>` in the diagram below. Each stack node consists of the stack frame constructor and four arguments. The fourth argument points to the original node in the heap representing a suspended call, in which one of the fields is being evaluated. The first argument is an unboxed argument which specifies the position within that node of an argument which needs to be evaluated before the execution of the call can continue. The second argument chains the stack frames together; it points to the next frame. The third argument is a function that acts as a continuation function. This continuation function will replace the one in the original node, once evaluation of all the fields in the pattern is complete.




The code of `strict011` causes a new stack frame representing the suspended call to be created on the stack. Its unboxed argument `3` is a binary number and indicates that argument `x` is evaluated. Its references are the previous stack frame, `old_st`, the continuation function `f'` and the suspended call `f x y` pointed to by the node `p`.

When an argument is evaluated, a return occurs. This puts the current node into the appropriate argument position in the suspended call and makes the next argument the current node. The pattern argument of the stack frame changes accordingly to indicate which argument is under evaluation. In this case, the pattern is now 2 to indicate that the other argument is under evaluation.

| `Current Node` | `Stack` | `Heap` |
|---|---|---|
| $< x' >$ | $st$ | $h \left[ \begin{array}{l} cn \mapsto < x' > \\ p \mapsto < f, x, y > \\ st \mapsto < fr, 3, old\_st, f', p > \end{array} \right]$ |
| $< y >$ | $st'$ | $h' \left[ \begin{array}{l} st' \mapsto < fr, 2, old\_st, f', p > \\ p \mapsto < f, x', y > \end{array} \right]$ |

When the last argument is evaluated, the appropriate argument position of the suspended call is updated, the info pointer of the suspended call is made to point to the continuation function, the suspended call is popped off the stack and is made the current node.

| `Current Node` | `Stack` | `Heap` |
|---|---|---|
| $< y' >$ | $st$ | $h \left[ \begin{array}{l} p \mapsto < f', x', y > \\ st \mapsto < fr, 2, old\_st, f', p > \end{array} \right]$ |
| $< f', x', y' >$ | $old\_st$ | $h' \left[ \begin{array}{l} cn \mapsto < f', x', y' > \\ f' \mapsto < ..., C_{f'} > \end{array} \right]$ |

### 8.6.2 Partial Applications

Partial applications also are introduced at compile-time, freeing the run-time system from the burden of run-time checks whether enough arguments are provided in a function call. There is a family of functions that deal with partial applications, one for each possible number of arguments.

As an example, the functionality of the built-in `pap1of2` is illustrated. This builds a partial application of a function of two arguments that is applied to only one argument. In the diagram below, `pap b` is a function of two arguments applied only to one. The code of its info node is $C_{pap1of2}$. Its references are a

function `f` of arity two as well as the second argument that needs to be supplied to `f`.

| Current Node | Stack | Heap |
|---|---|---|
| $<pap, b>$ | $st$ | $h \left[ \begin{array}{l} cn \mapsto <pap, b> \\ pap \mapsto <..., C_{pap1of2}, f, a> \end{array} \right]$ |
| $<f, a, b>$ | $st$ | $h \left[ cn \mapsto <f, a, b> \right]$ |

This can be generalised for any partial application `papnofk` of a function of `k` arguments applied to `n` of them, where `n` $\leq$ `k`.

## 8.7 Optimisations

More work is needed to investigate the efficiency of the Brisk machine, particularly in comparison to the `STG` Machine. Almost all of the optimisations which can be carried out before reaching the `STG` language (or from `STG` to `STG`) are also applicable in the Brisk setting; however, they have not been implemented in Brisk, making direct comparisons difficult at present. In particular, we have yet to investigate the implementation of unboxing. Optimisations which come after reaching the `STG` language, i.e. during code generation are more difficult to incorporate since they involve stacks, registers, return conventions etc.

Nevertheless, some of these have been investigated to see if similar optimisations are possible in Brisk. For example, in the `STG` approach, intermediate expressions are stored implicitly on stacks, which considerably reduces heap usage. In Brisk, intermediate expressions are stored as heap nodes. On the other hand, by treating the last node in the heap as a temporary node which can usually be overwritten by a node representing the next intermediate expression, most of the gain of using stacks can be obtained without complicating Brisk's simple approach to representation.

**Unboxing** The notation for strict types can be used for unboxing as well as for expressing restrictions in evaluation ordering, in essentially the same manner as in the Glasgow compiler. For each strict type, a decision can be made as to how values of that type are represented, as a number of words holding raw data

and/or a number of words pointing to heap nodes. Then, for each function, it is known what pattern of raw and pointer words to expect in the arguments to every call. The code generator can arrange for these to be shuffled so that the raw words occur before the pointer words, allowing a relatively simple and uniform run-time representation. If the `Int` type is defined by:

```
data Int = MkInt !Int
```

and !Int is unboxed, (!+) can be defined by:

```
(!+) :: !(!Int -> !Int -> Int)
x !+ y = let z = x !+! y in MkInt z


(!+!) :: !(!Int -> !Int -> !Int)
x !+! y = primitive ...
```

The function (!+!) adds unboxed integers and can be replaced by inline code. It violates the restriction that functions should not return strict types, but we can relax the restriction for primitive functions which are inlined.

As another example, imagine a function `g` which returns an unboxed type:

```
g :: Int -> !Int
```

This can be used in a valid way by inlining it as follows:

```
h = ...
    let n = g m
        ...
in ...
```

**Compile-Time partial Applications vs Run-Time Checks** An interesting question is to compare the Brisk machine approach to partial applications with the `STG`-machine approach to run-time checks. On the one hand, Brisk avoids the overheads of run-time argument testing and possible subsequent building of partial application nodes. On the other hand, unevaluated functions have to be evaluated before being called. The tradeoffs are not clear.

# Chapter 9

# The Brisk Run-Time System

This chapter describes the run-time system as developed for the Brisk compiler. It is especially designed so that it supports the properties of the Brisk machine, such as simplicity, flexibility, dynamic aspects and extensibility. One of its main characteristics is its ability to load functions as newly created nodes in the heap and to mix compiled and interpreted bytecode on a function-by-function basis. Several compilation techniques and design issues that have been employed to achieve the above mentioned goals are analysed.

## 9.1 Introduction

In the introduction of the new `GHC/Hugs` run-time system [95], Peyton Jones writes:

> *Compilers get the headlines, but behind every great compiler lies a great runtime system. A surprising amount of work goes into run-time systems, but disproportionately few papers are written about them.*

This chapter describes some aspects of the run-time system as developed for the Brisk compiler. [1]

It is especially designed so that:

- it supports dynamic loading and linking where functions are loaded as newly created nodes in the heap. This facilitates distribution and computational mobility where code can be communicated between processors and loaded up in the running environment.
- compiled code and interpreted bytecodes can be mixed on a function-by-function basis.

### 9.1.1 Dynamic Loading and Linking

Compilers translate source procedures to object modules. Then, the loader, or linker assembles all separately translated procedures to create the executable binary program. Traditionally, compilers would complete all linking before execution starts; this is called *static linking*. On the other hand, static linking does not fit well with virtual memory, as some procedures might me invoked only under special circumstances. `Dynamic linking` allows to link procedures at run-time, only when they are called. Current trends in operating system design

[1] The Brisk run-time system is joint work with Dr. Ian Holyer. A great deal of work in the code generator and front-end aspects has been made by Antony Bowers. Also, Chris Dornan implemented the type system and front-end aspects of a previous version of the compiler. Bytecode compilation, concurrency, distribution (distributed execution, dynamic loading/unloading of heap fragments in messages) and all the necessary design considerations and adaptations of the run-time system to support them, are part of this thesis. The Brisk garbage collector is described elsewhere.

favour dynamic loading and linking. First it has been pioneered by MULTICS but also the Windows operating system (all versions of Windows operating system, including NT) and UNIX followed. For a more detail description see also Tanenbaum [122]. In Windows operating system, dynamic loading is supported by special libraries called Dynamic Linking Libraries (DLLs), which allow modules to share procedures or data they contain. In UNIX the role of DLLs is played by special libraries called shared libraries. Following operating system design, dynamic loading and linking became also a major issue in compiler construction. The ability to load modules or code dynamically into the running environment is one of the current trends in compiler design. As a consequence, the design of the Brisk machine and the construction of Brisk compiler, has been strongly driven from the urge to provide a Haskell compiler that supports dynamic loading and linking.

Moreover, from the functional programming languages point of view, dynamic loading offers also additional important advantages. The purely functional operating system based on deterministic concurrency, as described in Section 5.5.4, together with dynamic loading of modules can be viewed as a single persistent program, where compiling and running a new module is equivalent to creating new functions and data types dynamically and then loading them up dynamically into the already running program.

Additionally, in a distributed setting based on deterministic concurrency, dynamic loading of graph fragments together with a scheme for unique naming of objects in the presence of distribution are the key issues to facilitate computational mobility. As referential transparency is preserved, computation can be moved on demand and the location of computation can change at run-time.

## 9.2 Code Generation

After a module is translated into BKL, it is passed to the code generator. This generates code either via C or via an interpreted bytecode. When a Brisk module is compiled via C, the code generator produces both a C file (.c) containing the relevant C procedures (see also Section 9.2.3) and a B file (.b) that represents the compiled Brisk module itself (see also Section 9.2.2). Such a compiled Brisk

module consists of a number of declarative entities, using a contiguous block of identifiers the purpose of which is to load function nodes dynamically in the heap. Declarative entities can be:

- functions;
- constructors;
- types;
- classes;
- overloaded functions;
- instances of overloaded functions;
- modules;
- any other entities;

For top-level externally visible declarative entities, i.e. entities that are exported from the module, a one-word Brisk identifier `BID` is allocated. Each `BID` is unique in a particular module environment and is allocated when the module is compiled. Non-globally visible entities are represented as relative addresses from the start of the block, see also Figure 9.1(a).

Also in Brisk, a `CAF` or constant-applicative form, i.e. a zero-argument definition such as `main = e`, is represented as a call to a zero-argument function, e.g. `main = main' ()`.

Compilation via interpreted bytecode produces only a `B` file. The difference between a `B` file produced from compilation via `C` and one produced from compilation via interpreted bytecode is that, in the latter, at the end of each function and constructor node one extra reference has been added that contains the bytecode string, see also Figure 9.1(b).

Also code for the special built-in functions together with their entity identifiers is defined in special core modules. Examples of such special core modules are `PreludeRun.hs` that represents the Haskell-definable aspects of the Brisk run-time system, `PreludeInt.hs` that provides built-ins to handle integers and arithmetic, `PreludeDist.hs` for distributed execution, `PreludeIO.hs` for `I/O`

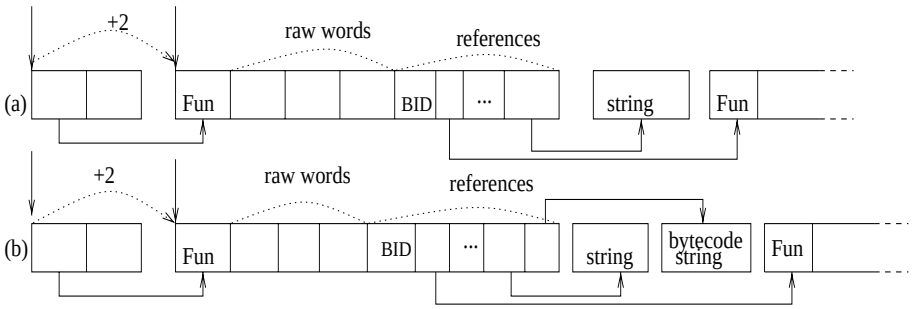



Figure 9.1: B File Layout

etc. The code from these core modules is intended to be used by the code generator rather than by programmers.

### 9.2.1 The Environment File

During code generation, an environment file is also created that represents a particular module environment. This contains an entry for each available module. Each entry consists of a `B` or `C` letter to indicate which kind of module it is (i.e a `B` or `C` file), the range of declarative entity identifiers that the module exports for `B` files or the range of code identifiers for `C` files and the module name. For `B` module entries, there is also a list of modules, from which the particular module imports entities. An example environment file is:

```
B 55 56 Main1 [ PreludeIO PreludeInt ]
C 91 95 Main1
B 46 54 PreludeIO [ PreludeC PreludeInt PreludeRun ]
C 56 90 PreludeIO
B 42 45 PreludeInt [ PreludeC PreludeRun ]
C 48 55 PreludeInt
B 41 41 PreludeC [ ]
C 47 47 PreludeC
B 0 40 PreludeRun [ ]
C 0 46 PreludeRun
```

For example, the first line indicates a `B` module that exports one globally known entity with `BID` 56 (`BID` 55 represents the module itself), the module name is

`Main1` and it imports `PreludeIO` and `PreludeInt`[2]. The environment file is used by the the module manager when loading modules dynamically in the heap.

### 9.2.2 B Module Files

A `B` file starts with a header. The header contains information such as whether the format of the words in the remainder of the file represents 32 bit words in bid-endian or little-endian format. The endianness of the module is checked against the endianness of the processor it is loaded and may engage reformatting, of string nodes or bytecodes, if necessary. The header is followed by a word giving the size of the rest of the file. The rest of the file is an array of words forming templates for a collection of nodes in the heap. The first node in a `B` file is a heap node that represents the module itself, called *module node*, see also Figure 9.4(a). It contains the range of globally known declarative entity identifiers of the module (i.e. 55-56 for `Main1`) and a pointer to an entity table which is indexed by this range and contains pointers to globally known entities relatively to the start of the array of words in the `B` file. In keeping with the design of the Brisk machine, each node template in a `B` file contains an info pointer, a number of raw words and a number of references. Both the info pointer and reference words can either be addresses relative to the start of the array, or globally known declarative entity identifiers. In the first case, the top bit is set. The info pointer word is the relative address, or entity identifier of an info node (inode) which describes other nodes and is needed during loading because it provides layout information for the node it describes. Also, a raw word may be an absolute value such as:

- the size, i.e. the number of words in a node;
- type, which specifies whether an info node describes expressions, simple function nodes, function nodes with a code word in them, or constructor nodes with an extra constructor word in them;
- shape, i.e number of raw words in a node;
- kind, i.e. with type information;

[2] `PreludeIO`, `PreludeInt` etc contain built-in functions.

- a code identifier representing a pointer to a `C` function.

If the `B` file has been produced by compiling via interpreted bytecode, info nodes that represent functions and constructor nodes are added an extra reference at the end that holds the bytecode string, see also Figure 9.1(b).

### 9.2.3 C Module Files

Compilation via `C` produces a `C` file in addition to a `B` file, containing the relevant `C` procedures. A `C` file consists of a collection of procedures using a contiguous block of code identifiers with the first representing the module itself, an array of their code entry points and a `C` module structure, as below:

```
#define F91 0 /* Main */
...

static void F94(...) { ... } /* main */
static void F95(...) { ... } /* function */
...
static C procedures[] = {F91,F92,...F95};
CModule M91 = {91, 95, procedures, "Main"};
```

Each `C` procedure in a `C` module similarly has a one-word code identifier `CID`, which is unique in a particular module environment and is allocated when the module is compiled. For example, the `CID` of the procedure `main` is 94 as it can be seen above. The only exported item is the `CModule` structure which contains:

- the range of code identifiers used;
- the module name;
- a pointer to the array of code entry points.

The exported name of the `CModule` structure is formed from the letter 'M' followed by the code identifier of the module. This allows, in principle, different versions of the same module to be linked into the same program.

```
typedef struct CModule {
```

```
    cid firstid;
    cid lastid;
    C *procedures;
    char *name;
} CModule;
```

An entry in the array of code entry points may refer to code of a:

- procedure in the `C` file;
- procedure in the run-time system;
- procedure in a library linked in with the program.

The loader makes no assumptions about what arguments are passed to the procedures. The `CModule` structure is copied into a suitable heap node.

## 9.3 Design Considerations

The first design consideration concerns the machine state. Brisk supports a number of execution models each of which might require additional registers, which could lead to a machine state with a huge number of registers. To avoid this, only a standard set of registers is defined in the machine state and, instead of defining additional registers, the root node feature of the Brisk machine has been exploited. The root node, also described in Section 8.5.1, is an array of node pointers that represents the current state of the heap and all active nodes are accessible from it, so that it can be used during garbage collection. This root node has been also used to store nodes of a particular interest. Each run-time system module can allocate to the root node permanent nodes of particular interest and access them using their indexes in the root node as identifiers, see also Figure 9.2. Thus, in the Brisk run-time system, a machine state consists of two parts:

- a structure that contains a number of registers, see also Section 9.7;
- special nodes, e.g. the scheduler, loader etc. held in the root node.

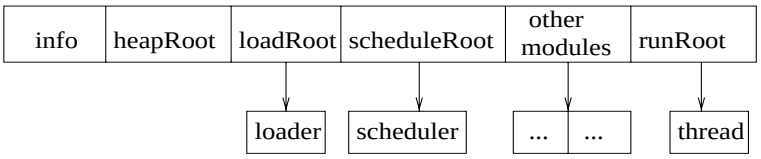



Figure 9.2: The Root Node

The next design consideration has been to facilitate dynamic loading. Execution code (generated C code or interpreted bytecode) is not allowed to access directly procedures or link with globals in the run-time system. To enforce this, items in the header files of the run-time system modules are protected by special conditional compilation flags, so that can only be included by other run-time system modules and not by execution code. Execution code is allowed to access functions in the run-time system only indirectly. There are two ways for doing this:

- through special macros defined in the header modules that transfer control of execution to the run-time system;
- by including pointers to special procedures as registers at the machine state.

The former technique is used when the execution code requires to alter the machine state. In this case, the execution code can communicate items to procedures in the run-time system using the thread register, as explained in Section 9.6.

The latter is used when a run-time system procedure needs to be accessed without altering the machine state or when a macro results to be impractical. This technique has been borrowed from the Java implementation and has been mainly used for distributed execution, described in Chapter 11.

Another design issue that facilitates dynamic loading is that the machine state registers are not passed as global variables to procedures. Instead, they are kept in a structure which is passed to the execution code as a pointer argument, see also Section 9.7.

The run-time system has four main component modules:

- The Heap Module;
- The Module Manager Module or Loader Module;
- The Scheduler Module;
- The Run Module.

## 9.4 The Heap Module

The heap module provides support for heaps with variable size nodes. A heap is represented as a structure that describes an array of words.

```
typedef struct heap {
   P base, free, limit, end; I id; P roots;
} heap;
```

The `base` and `end` pointers point to the beginning and end of the heap and the `free` and `limit` ones point to the beginning and end of the free space. Each heap has an integer identifier `id`, allowing to distinguish between several heaps, e.g. when simulating distribution. The root node, `roots`, is an array of node pointers from which all active nodes are accessible and it is used during garbage collection. The heap must be in a self-consistent state, with all active nodes accessible from the root node at any point where a garbage collection may occur. The heap operation `checkHeap` ensures that enough space is available in the heap so that new nodes can be allocated. If the execution code requires new nodes to be allocated, this can happen via specific macros defined in the run module, see Section 9.7.

To facilitate distributed execution by allowing to manipulate several heaps at once, procedures in the run-time system that operate on the heap take the heap as argument. Also, to facilitate dynamic loading heap operations are prevented from been called by execution code.

Facilities for allocating new nodes are provided. A heap node consists of an info pointer, a number of raw words and a number of pointer words. The info pointer points to an info node that describes the node by giving information about its type, size, i.e. the number of words in the node and shape, i.e. the

number of raw words in the node. This layout information is needed during loading. Also each info node has its own info node. The type field has a special indication for describing either non info nodes, info nodes with no further special fields, info nodes with one further special field containing a pointer to a `C` procedure or info nodes which also have a further special 'kind' word containing further type information.

Flexible arrays of node pointers (flex nodes), have been provided. *Flex* nodes store flexible arrays of node pointers, allocated backwards from the end of the node. Such nodes are used when the number of pointers that are going to be stored in the node is not known in advance e.g. when scanning through a node during message construction in distributed execution to record the pointers to non-global declarative entities. A flex node can be indexed, an item can be pushed on the end or popped off the end. The capacity of a flex node can be checked and the node can be resized if necessary. Also flexible arrays for storing raw data (*RawFlex* nodes) have been provided, with similar functionality and properties to flex nodes.

## 9.5 The Module Manager Module

The module manager (or loader) provides support for loading and linking modules dynamically. It is based on the loader node stored in the root node. The loader node contains references to two tables, the `B` module table that records information about `B` files and the `C` module table that records information about `C` files, as can be seen in Figure 9.3. Information stored in both tables is obtained by the environment file produced during code generation. Each entry in a `B` module table records:

- the range of globally known declarative entity identifiers (`BIDs`) of a particular module. This information is obtained from the environment file, created during code generation;
- a pointer to an entity table implemented as a flexible node (array of node pointers) so that it can be indexed by the range of exported `BIDs`. As modules are loaded dynamically, this entry is filled during dynamic loading, described in Section 9.5.1;

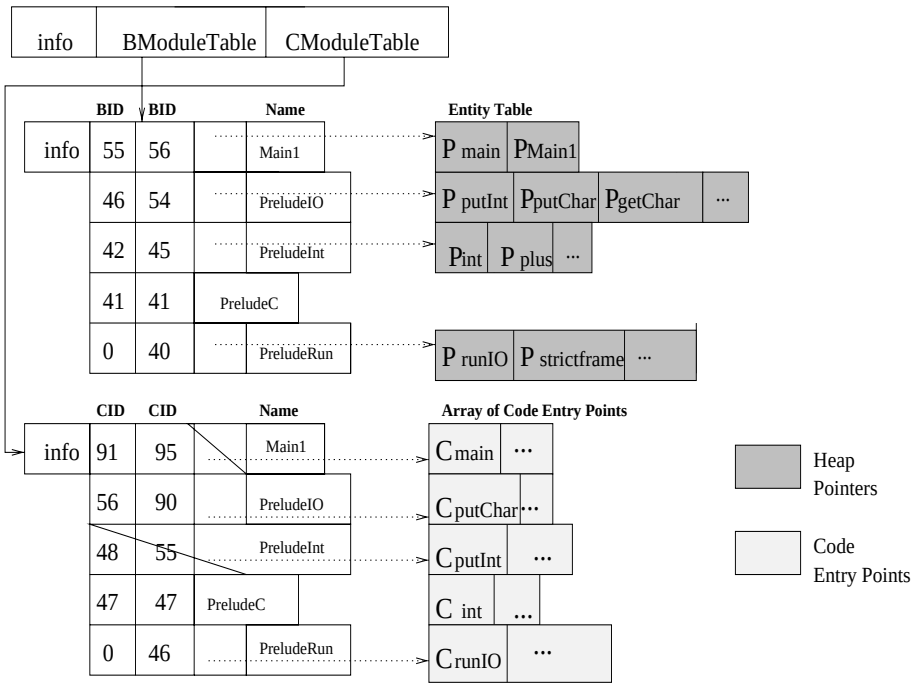



Figure 9.3: The Loader Node

- the name of the module.

The range of globally known declarative entity identifiers and the name of the module are filled before dynamic loading when the loader is initialised, whereas the pointer to the entity table is filled during dynamic loading.

Similarly, each entry in a `C` module table records:

- the first and the last code identifiers;
- a pointer to an array of the code entry points of the `C` functions implemented as a flexible node. An entry in this array may refer to code of:
  - a procedure in the `C` file itself;
  - a standard procedure in the run-time system;
  - a procedure in a library linked in with the program.

  At the moment, `C` functions are linked statically using information provided by the `C` module file produced during code generation;

- the name of the module.

### 9.5.1 Dynamic Loading

Dynamic loading of module templates (`B` files) is performed in two passes. In the first pass, a `B` file is read into a node in the heap, called block. This is a self-descriptive node that can hold module templates as a single block of raw data.

```
block = addBfile (h, block, mid);
  wrapper = BLOCKSIZE;
  while (wrapper < getISize (block)) {
     block = scanTemplate (h, block, wrapper);
     ip = ((P)block + wrapper);
     wrapper = wrapper + getISize (ip);
  }
...
```

The block is then scanned word by word to find out what other unloaded modules it references. These modules are also read into the block and scanned, until all relevant module templates are loaded. During scanning, if a word is a reference (pointer), it is checked whether this is a relative address or an declarative entity identifier.

If a reference is a relative address in the block, it is further checked to see if it is the first word of a node, i.e. an info node. If so, layout information is extracted (number of raw words and number references), the raw words are skipped by advancing the pointer accordingly and the number of references is extracted. If not, the number of references is decremented and the pointer is advanced to the next position. When the number of references becomes 0 then this is the info node of the next node in the module template.

```
if (ISSET(TOPBIT1,*p)) {
   if (nrefs == 0) {
      ip = ((P)block + template + (I)CLR(TOPBIT1,*p));
      i = i + 1 + getIShape (ip);
      nrefs = getISize (ip) - 1 - getIShape (ip);
```

```
        } else {
           i++;
           nrefs--;
        }
        continue;
     }
```

If the reference is an global entity identifier, i.e. a `BID`, the `BID` of its module is found using the range of the declarative entity identifiers recorded in the `B` module table. If the module is neither loaded in the heap nor read in the block of templates, this module is added at the end of the block and a new block is returned. Again, if the reference is the first word of a node, layout information is obtained from the `B` module table (if the module is already installed) or from the module node (if the module is in the block of templates), the raw words are skipped by advancing the pointer accordingly and the number of references is extracted. If not, the number of references is decremented and the pointer is advanced to the next position. When the number of references becomes 0 then this is the info node of the next node in the module template. The scanning continues until all the words in the block have been scanned and all the relevant module templates added.

```
        mp = findBModule (h, (bid)*p);
        mid = getBModuleId (mp);
        if (! isLoadedB (mp)) {
           tp = findTemplate (block, mid);
           if (tp == NULL) {
           newwrapper = getISize (block);
           block = addBfile (h, block, mid);
           tp = ((P)block + newwrapper + WRAPPERSIZE);
           }
        }
        if (nrefs == 0) {
           if (isLoadedB (mp)) {
              entities = getBModuleEntities (mp);
              ip = indexFlex (entities, ((bid)*p) - mid);
```

```
        } else {
         entities = tp+CLR(TOPBIT1,getBModuleEntities (tp));
         ip=tp+CLR(TOPBIT1,indexFlex(entities,((bid)*p)-mid));
        }
        i = i + 1 + getIShape (ip);
        nrefs = getISize (ip) - 1 - getIShape (ip);
     } else {
           i++;
           nrefs--;
     }
    }
return block;
```

In the second pass, the modules are linked and installed into the module table.

```
wrapper = BLOCKSIZE;
   while (wrapper < getISize (block)) {
      linkTemplate (h, block, wrapper);
      ip = ((P)block + wrapper);
      wrapper = wrapper + getISize (ip);
   }
```

During linking, the modules are scanned again word by word and that every reference is translated into a pointer. If a reference is a relative address, it is translated into a heap pointer.

```
if (ISSET(TOPBIT1,*p)) {
       *p = (P)block + template + CLR(TOPBIT1,*p);
}
```

If a reference is a global declarative entity, its module is found either already installed or in the block of templates, and the reference is translated into a heap pointer.

```
if (isLoadedB (mp)) {
    *p = indexFlex (getBModuleEntities (mp), (bid)*p - mid);
```

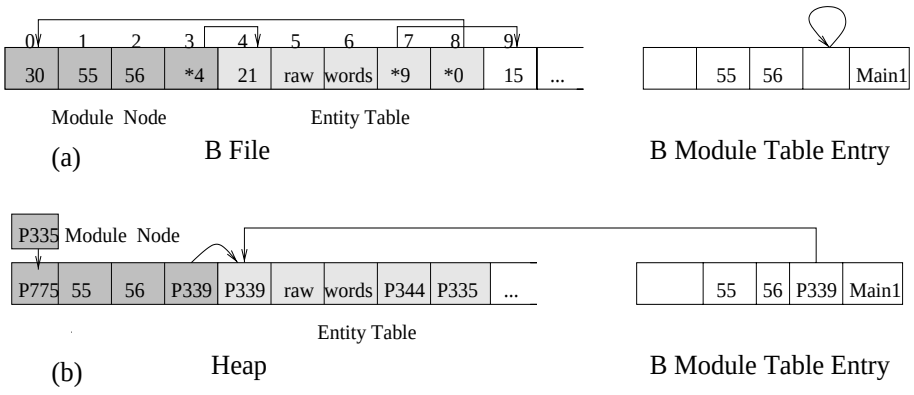



Figure 9.4: Dynamic Loading

```
} else {
    tp = findTemplate (block, mid);
    entities = getBModuleEntities (tp);
    if (ISSET(TOPBIT1,entities)) {
       entities = tp + CLR(TOPBIT1,entities);
    }
    *p = indexFlex (entities, ((bid)*p) - mid);
    if (ISSET(TOPBIT1,*p)) *p = tp + CLR(TOPBIT1,*p);
}
```

If the reference is the first word of a node, then the type field of the node is checked to see if it has an indication of a `C` code pointer. If so, the code identifier is extracted, the `C` module table is searched to find the module that contains the code with this particular code identifier and the code is extracted and linked. If the code identifier is 0, this is an indication that the module has been compiled via interpreted bytecode and the code of `Finterpret`, a special procedure that deals with interpreted execution in the run-time system, is linked.

The module templates are then installed in the `B` module table. As already mentioned, the first node in a `B` file is called module node and represents the module itself. It contains the range of global entity identifiers used in the module and a pointer to an entity table that holds pointers to those entities relative to the `B` file, see also Figure 9.4(a). After linking, the entity table holds heap pointers that point to those global entity identifiers. During installation, each

entity table entry in the `B` module table (initialised as empty) is updated to contain a pointer to the entity table of the module node, see also Figure 9.4(b).

The linking scheme as described is rather immediate, in the sense that node templates can immediately become normal heap nodes. It can though be easily adapted to be more dynamic. Instead of representing an unlinked pointer as an entity identifier, we can represent it as the relative address of an import node. This contains an entity identifier as a raw word, and turns itself into an indirection when executed, but we do not pursue this further here.

### 9.5.2 The Interpreted Machine Model

**Bytecode Compilation** During bytecode compilation, interpreted code is generated by translating `BKL` constructs into Brisk abstract machine instructions `BAM`. These have been already explained in Section 8.4. `BAM` instructions may have either one or two fields which leads to compact bytecode. The first field of every instruction is the `opcode` (`operation code`) which identifies the instruction. The second field, for those instructions which have one, specifies the `operand`, e.g. `opLocal n`, to push a copy of the `n`'th item on the top of the frame. Both interpreter opcodes and their arguments are represented as a list of non-negative integers, see also Figure 9.6. To allow a compact bytecode encoding, opcodes 0 to `15` have no arguments and the rest have one argument. Opcodes `0-29` are represented most compactly. Opcodes `30` onwards, if used, are represented using an extension mechanism.

The interpreted bytecode is then translated from a list of integers into a list of bytes by compressing two nibbles into one byte, see also Figure 9.5. If the first nibble is `0`, the second is an opcode from `0` to `15`. If the first nibble is `1`, this represents an extension mechanism, where the second nibble indicates the number of following bytes containing the opcode and the argument. If the first nibble is `2` to `15`, this represents an opcode from `16` to `29` and the second nibble represents the argument. If the argument is more than `15`, then the second nibble is `15` and the actual argument is in the next byte. If the argument is more greater `255`, then the general extension mechanism is used instead.

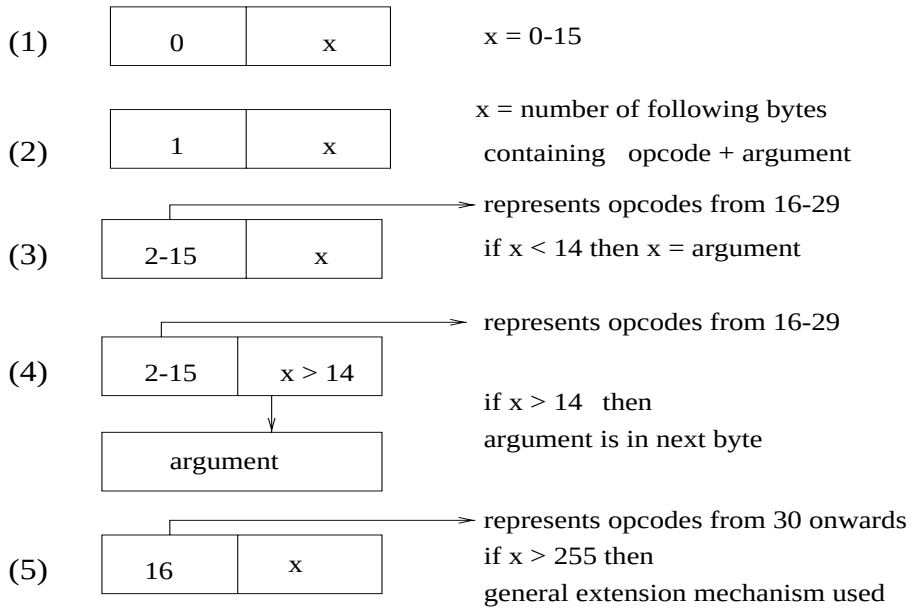



Figure 9.5: The Bytecode Compaction Mechanism

**Bytecode Execution** During dynamic loading, the nodes that contain pointers to code identifiers link their corresponding code into the loader. If the code identifier is 0, this is an indication that the module has been compiled via interpreted bytecode and a special function, `Finterpret`, responsible for interpreted execution is invoked. This creates a *frame* of local variables used during execution, also explained in Chapter 8.4. The frame is local to each individual function call and starts out empty. It is extended in a stack-like manner as needed. This stack-like use is very similar to the stack in the original `G`-machine and allows the `BAM` instructions to have at most one argument each, often a small integer, which leads to a compact bytecode. It is implemented as an array `V` of local variables, dynamically resized when necessary. During execution, an index `NV` is kept pointing to the next unused slot in the array. For case switches, a dynamic array `T` is used as a jump table and similarly a pointer `NT` is used as an index to the next table slot.

The interpreter uses a `256`-way switch to decide what to do with each byte, for maximum speed. Although the bytecode format is capable of dealing with arguments up to `8` bytes, there are unlikely to be more than `4` at present because

| | |
|---|---|
| *ENTER* | *Make the node on top of the frame the current node* |
| *RETURN* | *Return the current node into the appropriate argument position of the previous node and pop the return stack* |
| *FRAME n* | *Declare a frame of n local variables* |
| *GETLOCAL n* | *Push a copy of the n'th item on the top of the frame* |
| *CONST n* | *Push a constant onto the heap* |
| *REFS n* | *Load n global references into local variables* |
| *ARGS n* | *Load n arguments into local variables* |
| *MKAP n* | *Build a node of size n* |
| *ALLOC n* | *Allocate space for new heap node in next variable* |
| *FILL n* | *Fill a previously allocated node* |
| *TABLE n* | *Allocate a jump table of size n* |
| *CASE n* | *Set next jump table entry point to current location* |
| *DEFAULT n* | *Set default entry point to current location* |
| *SKIP n* | *Skip a number of instructions* |
| *SPLIT n* | *Extracts n fields from the case expression* |
| *SWITCH n* | *Switch on the constructor in the current node* |
| *RESERVE n* | *Reserve n words of heap space* |

Figure 9.6: The BAM Instruction Set

of the limitations on the Haskell Int (and C int) types. The code executed by each of the operations is defined in the run-time system.

## 9.6 The Scheduler Module

The scheduler module is designed to support multi-threading. The Brisk run-time system supports a round-robin scheduling scheme. This is based on a scheduler, see also Figure 9.7, which is a heap node that maintains a collection of runnable threads. The scheduler node is stored in the root node. The scheduler node contains 7 fields, see also Figure 9.7(b):

- an info node;

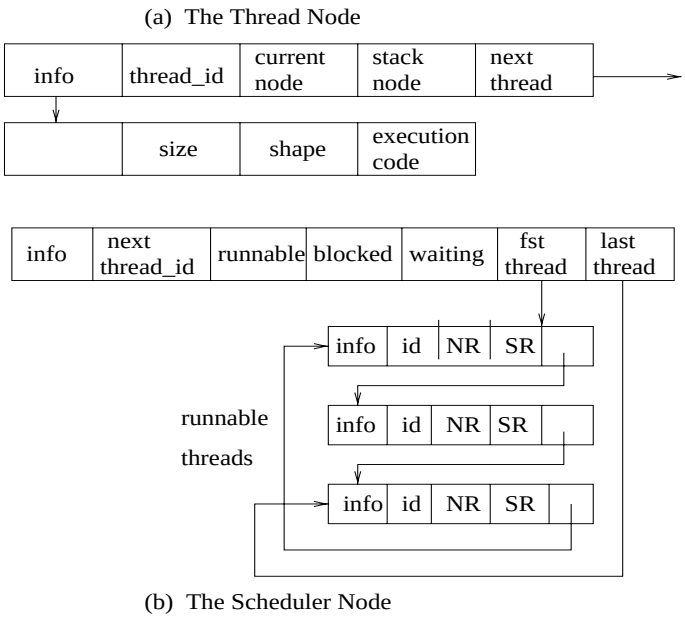



Figure 9.7: The Scheduler Node and The Thread Node

- a raw argument holding the next thread identifier to be assigned to the next created thread;
- counters for the runnable, blocked and waiting threads respectively;
- pointers to the first and the last item of a circular queue of threads.

Threads which block or wait are assumed to have been queued elsewhere; the scheduler only records how many there are.

A scheduler is initialised with zero runnable, blocked and waiting threads and null pointers for both the first and the last circular queue of threads. The scheduler has no information about the execution model of each thread; each thread carries its own execution code in its info node.

A number of functions that manipulate the runnable queue are also provided, such as `unblock` `unwait`, `spawn` etc. These update the counters of blocked, waiting or runnable threads and enqueue the relevant threads to the runnable queue. As already explained, these procedures are included within conditional compilation flags so that they can be accessed directly only by other run-time system modules but not by execution code. Execution code can access these

procedures only indirectly by calling a suitable macro that returns control of execution to the scheduler which can access them and perform the required operation.

A thread node has 5 fields, see also Figure 9.7(a):

- an info pointer;
- a thread identifier;
- a pointer to the current node register;
- a pointer to the stack register;
- a pointer to the next thread in the queue of runnable threads. When a thread dequeued from the scheduler to be executed, this field can be used for transferring items from the execution code to the run-time system, see also Figure 9.8.

The thread node may have further fields which point to additional registers that are particular to other execution models, as for supporting logic programming extensions. In addition, the info node of a thread node has an execution code entry point which runs a thread for one time-slice.

When a program runs, control is transfered to the scheduler. This calls repeatedly the execution code of each runnable thread for one time-slice, until they all finish.[3]

```
int schedule (heap *h)
{
...
/* dequeue the first runnable thread */
thread = dequeue (scheduler);
/* get its info node */
ip = getInfo (thread);
/* call its code from its info node */
call = (int (*)()) getICode (ip);
```

[3]Macro `getInfo(p)` returns the info node of `p`. Macro `getICode(p)` returns the code of node `p` accessible from its info node.

```
/* return a condition to the scheduler,
   when its time-slice finishes */
condition = call (h, thread);
switch condition
...
}
```

When a thread finishes its time-slice, it returns an integer condition to the scheduler. This integer specifies the condition of the running thread and determines what the scheduler should do with the thread. The condition of a thread can be either `RUNNING` with `n` execution steps left, or `RUNNABLE` when it is queued in the runnable queue, `BLOCKED` because some other thread has a required resource or is evaluating a shared node or in a distributed execution it waits for a reply message, `WAITING` for `I/O` on e.g. a channel or a timer, `SPAWING` when it is asking for a new thread to be created or `DYING` when its current node has reached weak head normal form. As we will see in Chapter 11, for distribution purposes the condition of a thread includes additional options which transfer control to distribution specific modules.

## 9.7 The Run Module

The run module holds part of the machine state as a structure that contains a number of registers. The other part of the machine state, i.e. the scheduler, and the loader nodes are held in the root node.

The registers include the current node, the stack, the thread, the free heap memory, the heap memory limit, the condition and the identifier of a particular machine registers. To support dynamic loading, the structure holding the machine state is passed to the execution code as a pointer argument.

```
#define BRISK_REGISTERS
  P N;    /* current node register */       \
  P S;      /* stack register */            \
  P T;      /* thread register */            \
  P H;      /* free heap memory register */  \
  P L;      /* heap memory limit register */ \
```

```
  I C;      /* condition register */         \
  I ID;     /* machine identifier */          \
  heap *h;   /* the heap structure */

typedef struct mc {
   BRISK_REGISTERS;
} mc;
```

A number of macros have been also defined which allow the manipulation of heap nodes, such as accessing references (`GLB(p, i)`) and arguments (`ARG(i)`) of a node, checking for heap space for allocating new nodes and filling them with given values, or making a node the current node.

Also, the macro `FIX()` has been defined for turning a temporary node into a permanent one. In the Brisk machine, see also Section 8, function applications correspond to tail calls, which consists of building new nodes. Although tail calls simplify the run-time system by avoiding the use of extra stacks to hold intermediate expressions, they require a large amount of heap space as they create a large number of nodes. To reduce the number of heap nodes, tail calls create temporary nodes rather than permanent ones. A *temporary* node is created by allocating a new node at the end of the heap, but without advancing the free space pointer. Such nodes get overwritten by new ones they create. Any node, the code of which assumes that the current node will continue to exist such as constructors, functions which save pointers to the current node in stack frames or elsewhere, or functions which reuse the current node as a space optimisation, must make themselves permanent by advancing the free space pointer before continuing execution.

### 9.7.1 Execution

**Initialisation** Execution starts by creating a new machine, i.e. a new heap together with a loader and a scheduler node. A program is loaded onto the heap, using its `B` module identifier `BID` allocated during code generation. A new thread is also created; its current node points to the expression `runIO main World`. The `runIO` function runs programs. It is strict in both arguments and when their evaluation finishes its execution code causes the code of the stack to

be executed and the result to be printed. The thread is then enqueued to the scheduler which takes over.

**Execution** The scheduler dequeues the first runnable thread and calls repeatedly its execution code until its time-slice finishes or the heap node pointed to by the current node register of the thread is evaluated to weak head normal form. If there are more than one threads, then they are executed in a round robin fashion each for one time-slice. The execution code of each thread loads the machine registers (using information from both the heap structure and the thread's registers) and repeatedly calls the execution code of the current node for one time-slice.

```
M = &machine;
/* load the registers */
M->H = h->free;
...
/*run the code of the current node
for a time-slice */
M->C = RUNNING(SLICE);
while (M->C < RUNNABLE) {
   M->C++;
   call = getICode (getInfo (M->N));
   call (M);
}
/* make current node permanent */
FIX();
/* restore the machine registers */
h->free = M->H;
...
/* return a condition to the scheduler */
return M->C;
```

When the time-slice of a thread finishes, the state of the machine is saved. Saving the state of the machine involves:

- making the current node permanent;

  As already explained, as a space optimisation the current node register may refer to a temporary node stored at the beginning of the free space instead of a permanent one so that it can be immediately overwritten without requiring garbage collection. When the state of the machine is saved, the temporary node to which the current node register points is assumed that will be needed further and so it must be saved and become permanent.

- restoring the values of its registers into the heap structure and thread.

Control is then returned back to the scheduler by returning a new condition to it, indicating what the scheduler should do with that thread. If the condition indicates that the thread's current node has not yet reached weak head normal form, the thread is enqueued back to the scheduler, otherwise the thread dies and is not enqueued back to it.

If during the execution of a thread, the execution code needs something special to happen before its time-slice finishes, e.g. if the execution code needs to create a new thread, it calls one of the macros defined in the header files of the run-time modules. This updates the condition register with a special value, which causes the machine to shut down and control is returned to the scheduler. As already explained, shutting down the machine involves making the current node permanent and saving the state of the machine. The scheduler enqueues the previously running thread and uses the new condition to access code in the run-time system to satisfy the request, e.g. creating a new thread. After satisfying the request, the scheduler dequeues the next thread and execution continues by calling its code, which in turn creates a new machine state and executes repeatedly the code of the current node.

As an example of how execution code can access indirectly run-time system functions, we illustrate how a new thread is spawned. The execution code calls a suitable macro `spawn(p)` defined in the header file of the scheduler module.

```
#define SPAWN(p) {putNextThread(T,p); C = SPAWNING;}
```

Macro `spawn(p)` causes node `p` to be saved in the last slot of the running thread, see also Figure 9.8. This slot is used for chaining with other threads when the

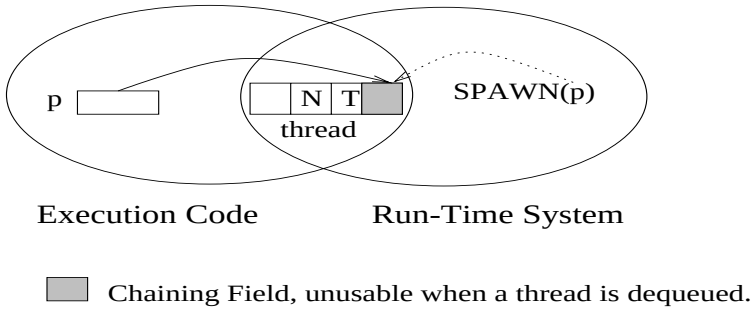



Figure 9.8: Passing Arguments from Execution Code to the RTS Functions

thread is enqueued to the scheduler. As the thread is current, i.e. runnable, it means it is dequeued from the scheduler and so, this field can be used to transfer items from the execution code to the run-time system. It also updates the condition returned to the scheduler to a special value. This causes the machine state to shut down and control to be transfered to the scheduler. The scheduler creates a new thread assigning the node `p` to be its current node and enqueues the previous running thread to the scheduler.

```
switch (condition) {
   case SPAWNING:
/* create a new thread with current node
the node p saved by the SPAWN(p) macro */
      spawn (h, getNextThread (thread));
/* enqueue the previous running thread
to the scheduler */
      enqueue (scheduler, thread);
      break;
}
```

The `spawn` function can be accessed only by the scheduler and not by execution code. It gets from the scheduler the next thread identifier, makes it the identifier of the new thread and enqueues the new thread to the scheduler.

```
void spawn (heap *h, P thread)
{
```

```
    P scheduler; I id;
    scheduler = getPtrWord (h->roots, scheduleRoot);
    id = getNextThreadId (scheduler);
    putThreadId (thread, id);
    putNextThreadId (scheduler, id+1);
    enqueue (scheduler, thread);
}
```

**Termination** If the current node of the thread has reached weak head normal form, the condition returned to the scheduler indicates that the thread dies and is not enqueued back to the scheduler. When the current node of the last thread (i.e. the thread which executes main) reaches weak head normal form, its stack node has a special code which arranges the result to be printed and the execution to terminate.

# Part IV

# Functional Distribution

# Chapter 10

# Distribution in a Demand Driven Style

In this chapter we present a model of distribution as a natural consequence of deterministic concurrency, a purely declarative form of concurrency that preserves referential transparency. This extends the demand driven model, used as the operational basis of functional languages, into one with multiple independent demands without the need to allow non-determinism to appear at the user level.

The abstract model of this distributed system uses a uniform access memory model in order to provide a global memory view. This, combined with explicit thread based concurrency and explicit representation of demand, enables the trustable distribution of reactive systems across different processors. Since referential transparency is preserved, any pattern of distribution has no effect on the semantics of the functional program. Data, code and computation can all be moved in response to demand. This provides both a conceptual and operational model for mobile computation. In computer science literature, computational mobility has at least two meanings. The ability to move computations is one; the ability to tolerate disconnection of one computer from another in an orderly manner is another. In this thesis we are dealing with the former. Program fragments can migrate from one processor to another under the control of program annotations.

## 10.1 Introduction

Distributed applications are becoming more and more pervasive since they play a major role in the current trends of computing and represent a significant proportion of computer software. Such applications include the following classes as identified by Bal [7, 87]:

- parallel and high-performance applications;
- fault-tolerant applications and real-time systems;
- distributed operating systems;
- inherently distributed applications such as email and World Wide Web applications.

In this chapter we focus on mechanisms to support inherently distributed applications. Such applications involve many independent components, possibly located at different places, which cooperate in order to provide a single integrated service and enable resources to be shared via a network. Also, the expansion of the Internet requires that distributed applications evolve further, allowing the mobility of applications while minimising the impact of hardware issues. Such mobile software applications aim at providing online services, interactive distance learning, home networking etc.

We present a new conceptual model for building distributed applications which is derived as a natural extension of deterministic concurrency, its most important characteristic being that it retains referential transparency and does not allow non-determinism to appear at the user level. The main virtue of this system is that applications that use it are trustable in the presence of distribution. This trustability derives from the fact that programs have repeatable and resilient behaviours, properties that emerge from the preservation of semantics through deterministic concurrency. Furthermore, this distributed system allows for mobile computation since it exhibits performance flexibility, i.e. it maximises the scope for evaluation independent of location and can naturally express distribution, migration and mobility.

## 10.2 Purely Functional Distributed Systems

### 10.2.1 Concurrency and Distribution

Distributed applications require concurrency in their underlying implementation to increase their expressiveness. For example, an Internet Web browser typically uses concurrency to create multiple connections to servers and to display items for the user while other activities are going on. As already explained in Chapter 2, since concurrency introduces non-determinism as a consequence of timing dependencies during process creation and communication, all existing distributed systems suffer from complexity and unpredictability. Moreover, non-determinism problems, which are rare on a single processor, can become more common in a distributed setting, because of the effect of network load. For example, even the common convention of "double-clicking" a mouse button, where two clicks which are close enough together in time are treated as having a special meaning, may not work over a remote connection to a computer if the times of the clicks are measured at the remote end. All this conspires to make distributed systems even more difficult to debug, adapt and maintain.

### 10.2.2 Deterministic Distribution

To address the issue of distribution, concurrency is a necessary prerequisite. Since we did not want to compromise referential transparency, we have investigated distribution as a natural extension of deterministic concurrency. Deterministic concurrency offers a demand driven approach to concurrency which extends the single demand execution model into a model with multiple independent demands. Concurrency primitives are provided which spawn independent threads of execution. As it has been explained in Chapter 6, this is achieved by *state splitting* [55], a technique which assigns each new thread part of the state, a *substate*, to act upon. Distinct concurrent threads are restricted at creation to access disjoint substates. Threads communicate only via shared values (not stateful ones), including streams which they take as arguments or produce as results [51]. Alternatively, threads communicate via a safe form of message passing with timed deterministic merging of messages from different sources. This is achieved by introducing a hierarchical notion of time based on the ordering of

communication events within the threads[56]. The use of timestamps on messages together with hidden synchronisation messages ensures, in a transparent way, that all merging of message streams in a program is deterministic, without any unnecessary delays.

To demonstrate the validity of such a form of concurrency, Carter [23] built a prototype implementation of a single-user multi-threading environment. In order to focus on the main ideas the following simplifications were undertaken:

- A fixed mapping of processes onto processors;
- Statically defined communications interfaces, i.e. processes communicated only via a statically defined network of streams;
- Separate local memories.

Within the semantic framework that has been developed for deterministic concurrency, distribution is almost an orthogonal issue. Preserving referential transparency through deterministic concurrency during thread creation and communication allows to split concurrent systems into collection of threads or processes so that the entire collection of processes is equivalent to a single heap program. Moreover, these threads can be distributed across a number of processors, see also Figure 5.5 without any change in the meaning of programs. Thus, systems can be distributed across a number of processors in a trustable way, without any change to the meaning of programs - the only observable difference being the speed at which the overall system generates results.

Based on the results of the above mentioned feasibility study, we removed its initial set of restrictions, allowing the pattern of processes and their interconnections to evolve, allowing code and computation as well as data to migrate among processors and having a global memory model to give such migration a clear meaning. This provided a conceptual framework which allows for arbitrary distribution of computation in a demand driven style [119], since the semantics of programs are preserved independently of how a program is distributed. It also offers a trustable form of distribution which ensures reliability and effective reuse of software. In other words, deterministic concurrency extends the call-by-need model of lazy evaluation into one with multiple independent demands and deterministic distribution enables these independent demands to reside in

separate processors. Additionally, it offers the flexibility to choose dynamically whether unevaluated code fragments or result values are communicated. This allows for new approaches in load balancing and issues of value security and integrity. In this framework all entities are mobile; computation can be moved dynamically on demand in an entirely arbitrary manner. This distributed model provides a number of important characteristics, most of them based on the fact that distribution is orthogonal to functionality:

- It allows to treat any mechanisms such as distribution mapping, processor identification, or granularity of the distribution purely as annotations to the program. An annotation is an explicit indication in the representation of a program which changes its operational semantics, but not its meaning. An annotation is semantically equivalent to the insertion of the identity function in the source representation, i.e. it is semantically inert and at the same time able to change aspects of the operational behaviour of a program. Annotations can be static, i.e. added to the program source text by the programmer, or dynamic, added by the run-time system according to resource measurements. Distributing the program becomes an operation that can be performed late in the development life-cycle and the programmer is freed from the burden of explicitly interweaving all the distribution details within program code.

- It makes a clear distinction between state, communication and functionality, which further allows the separation of the semantics and design of programs from their distribution. This is in sharp contrast to traditional distributed programming models [118] such as client-server and RPCs, where all the distribution mechanisms have to be programmed explicitly.

- This distributed model also provides a high level of trustability. This is guaranteed on the one hand by the deterministic concurrency and on the other by the fact that distribution is an issue orthogonal to the functionality of a program. Deterministic concurrency guarantees the correctness of a concurrent program in an one-processor setting. The fact that distribution is orthogonal to the design of the program ensures that no hidden operational bugs related to accidental timings arise in any pattern of dis-

tribution.

- Distributed programs can be easily debugged. A program can be developed on one processor, where reasoning and debugging are relatively easy before being distributed, i.e. one can test a program on one machine and run it over many machines. Proof of functional correctness apply equally to the program, whether running as a single processor system or as a distributed system.

- It provides an approach to distributed problem solving which enables the implementation of reactive systems, the behaviours of which remain predictable and are independent of how they are distributed. Referential transparency ensures that in a distributed setting, output values happen in the right order, since they are fully defined only by the sequence of input values.

## 10.3 The Operational Model

Functional programming languages are well known to have a close relationship between operational and denotational semantics. Our approach to distribution is to introduce operational activities in the form of annotations that have no effect on the declarative semantics.

The operational model for distribution uses a uniform access method in order to provide a distributed, transparent addressing scheme and to enable global knowledge of the current state of the system. This, combined with explicit thread based concurrency with no user exploitable non-determinism and explicit representation of external demand, provide the mechanism to distribute reactive systems on several processors and offers a new model for computational mobility. As we have totally preserved referential transparency, any pattern of distribution has no effect on the denotational semantics of the system.

This operational model uses referentially transparent mechanisms which are very similar to those used in parallel functional implementations [101, 64, 24, 126], but adapted to serve distribution purposes, making distribution referentially transparent. A conventional distributed heap mechanism is employed, in a

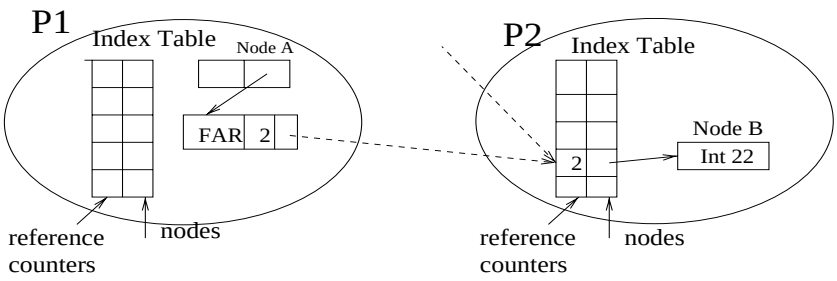



Figure 10.1: The Global Memory model

way that it supports distribution rather than parallel efficiency gains. Also conventional message passing techniques are used not as an explicit communication mechanism for programmers but to enable migration of computation transparently. The operational model has a number of important characteristics, most of them based on the fact that distribution is orthogonal to functionality:

**Global Memory Model** The operational model implements a global distributed heap. It consists of a collection of processors each of which contains heap allocated objects which form part of the overall graph as specified by annotations to the program. The complete set of heaps can be thought of as forming a single global memory which contains the complete graph of the program.

A remote pointer from a node A to a node B in separate processors is represented as shown in figure 10.1. The node A points to a remote indirection node, or *far node*, that acts as a local synonym for node B. Such a remote reference contains two pieces of information. The first one indicates which processor is the owner of the target node and the second specifies an offset in an index table in the remote processor. Each processor includes an index table which records those nodes which are referred to from outside of the processor. Each index table entry has a reference count and a pointer to the relevant local node. The reference count keeps track of the number of far nodes in the system which refer to the local node. The index table thus, decouples the local heap from the global distributed heap, providing a global address space where objects can be uniquely named and so, it records global knowledge of the inter-processor aspects of the system. This allows nodes to be moved in the local heap without affecting remote references and thus enables local garbage collection to be

carried out independently from the rest of the system.

Information is transferred from one processor to another using low level messages. When processor `P1` encounters a far node during execution, it needs to obtain the value of the remote node from processor `P2` which is the owner of the requested piece of graph. A new entry is added to `P1`'s index table, referring to the far node, which is locked to record the fact that the value is on the way and which acts as a target for the reply. A request message is sent to `P2` containing remote pointers to both the desired node and the original far node. The processor `P2` returns a copy of the requested node in a reply message to `P1` together with the reference to the original far node. When `P1` receives the reply message, the far node is over-written with the node obtained from `P2`. Both processors reduce their reference counts as appropriate to preserve the invariant that each reference count records the number of remote references, whether from far nodes or from messages in transit.

Variations on this mechanism allow a variety of policies to be implemented. Normally, a far node represents a value and the requested node is evaluated as necessary before being returned. An alternative type of far node represents a request for ownership; when the node is returned by `P2`, it is replaced by a far node in `P2` pointing to the new version held by `P1`. A third possibility, suitable for more permanent values, is to request a cached copy; `P1` retains the far node as well as its value so that later requests for the same permanent value can be detected and dealt with locally.

**Deterministic Concurrency** This is achieved through lightweight processes (threads) in each processor which share access to the program graph, but which have independent states. Threads communicate via shared non-state values, or via deterministic time-merged channels as described above. A thread which attempts to evaluate a piece of graph locks it during evaluation to prevent any other thread from duplicating the work of evaluation unnecessarily. When a particular piece of graph is evaluated, its value can be shared at will.

The difference from a concurrent sequential setting is that in a distributed setting (figure 10.2) the execution line moves dynamically from processor to processor resulting to a collection of execution lines into which threads alternate.

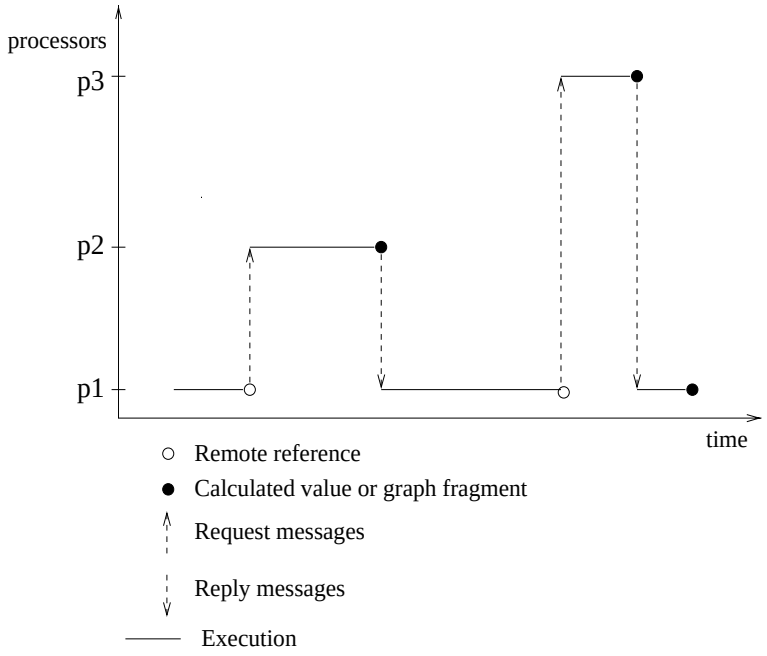



Figure 10.2: Distributed Execution

Any ordering of evaluation of those threads does the actual values produced. Messages transmit the control of execution remotely by carrying demand causing the evaluation to happen in a different execution line.

**Explicit Representation of Demand** Another feature of our distributed operational model is the explicit representation of demand to initiate remote graph reduction. In functional languages, evaluation is forced by the demand for the value of a graph node. This is also called call-by-need semantics. In a sequential program, there is a single source of demand which causes graph reduction. In a concurrent setting the graph is evaluated concurrently by allowing more than one source of demand. In a distributed setting, multiple sources of demand may reside in separate processors. Messages carry demand across processor boundaries; when a message requesting a node arrives at a processor, it causes a new thread to be created, which represents a new demand forcing a local piece of graph to be evaluated. Control of execution moves dynamically from processor to processor, driven by messages. From the point of view of

an external observer of the execution all we have introduced are differences in the relative timings of values generated by the overall system, not the values themselves or their order. Tracing the execution of a thread becomes equivalent to observing the passage of demand and computation through the distributed graph.

**Migration of Computation** This distributed system allows for variable granularity of distribution which is specified by explicit annotations at various points of the program graph to indicate which portions of it should be evaluated locally and which remotely. A single node which is moved can represent a simple value, or a large data structure, or an arbitrary amount of computation.

This provides the underlying operational infrastructure to support migration as a mechanism to satisfy system constraints, e.g. performance or security. Migration happens by allowing both flow of data and flow of demand through messages. Different kinds of demand can be supported, e.g. a piece of the execution graph can be evaluated by the owner and its value returned, or ownership of the unevaluated graph can be handed to the requester, so that the requester evaluates it. There are two bases for making this choice.

The first is a question of choosing the most appropriate site for evaluation. For example, in a web-based application, the client may ask for the result of a database query which the server evaluates, whereas it may ask for ownership of a graphics computation which it executes locally. Dynamically, the mechanism can be used for load balancing; if the owner of the subgraph is more heavily loaded than the requester, it is better to return the unevaluated subgraph and allow the requester to do the evaluation.

The second is a question of communication overhead. Evaluating a subgraph before returning it may result in fewer messages overall. In addition, expressions can be evaluated in advance before they are requested by introducing extra threads that correspond to additional demand. These threads may either be speculative, or justified by strictness information which shows that the expressions will be needed. This acts as a mechanism which tolerates latency. It means that values are available when needed and allows them to be sent together in bulk in the same message to reduce the number of messages.

**Highly Available Distribution** The global memory model of distribution can provide support for long-lived entities, i.e. data and programs that last for a complete user session, or are part of a permanently running operating system environment. A long lived data value belongs to a particular processor and its remote reference is used as a long-term identifier. Long-lived programs are threatened by long-term garbage. Cycles of distributed garbage entities which span processors lead to degradation of such programs, but this is notoriously difficult to deal with in interactive systems where it is not practical to bring the entire system to a halt in order to collect garbage.

We outline here a hierarchical, non-blocking, distributed garbage collection scheme which deals asynchronously with garbage. This allows for maximising garbage collection efficiency under the assumption that most garbage is locally defined, i.e. garbage collection is performed within the local area more frequently. This approach also allows the isolation of non-available portions of the memory while partial garbage collection occurs.

Collecting distributed non-cyclic garbage is based on a variant of Lester's distributed garbage collection algorithm [73]. The reachable set of nodes in the distributed graph as a whole involves both local pointers and remote references. The latter are recorded in the index table with their reference counts. If a reference count becomes zero, the index table entry is cleared and the corresponding node becomes local.

For collecting distributed cycles of garbage entities without bringing the system to a halt, we propose a scheme similar to the one proposed by Maheshwari and Liskov [77]. This scheme forces cycles of entities which might have not been touched for a long time to migrate into a single processor, where local garbage collection techniques can be used. These techniques deal with global cyclic garbage at a variable rate as needed, simultaneously with local garbage collection.

**Sharing** In keeping with the traditional functional paradigm, sharing ensures uniqueness of unevaluated objects in a distributed environment. This is achieved either by distributing references or by giving away a piece of graph and replacing its local copy with a reference. Operationally, sharing is preserved by the index

table through a reference counting scheme which records the number of remote references to a particular piece of graph. It also ensures that the distributed graph is consistent, i.e. equivalent to a non-distributed graph after performing the same number of steps. It also shows that a distributed program is equivalent to a program within a single processor, with the sole exception that not all of the unreachable memory (garbage) has been recovered. Evaluated objects, i.e. values, are infinitely sharable and can be freely copied.

**Performance Flexibility** Performance flexibility has two aspects in this distributed system. The first one is flexibility with respect to computational time. Any additional parallelism is orthogonal to the distribution process and can be introduced in the form of annotations to the program graph. Thus, evaluating expressions in advance before they are requested (either speculatively or because strictness information shows that they will be needed) reduces communication overhead. For example, when communicating a list, instead of using a separate forcing operation for each list element, the whole list or parts of the list can be evaluated and communicated.

Moreover, *pipes* can be introduced to increase efficiency, provided that it is guaranteed that they do not change the semantics. In Figure 10.3, processor `A` eventually requires a list from processor `B`. Instead of having a separate forcing operation for every element which would be too slow, a pipe can be introduced, by creating a speculative thread to satisfy this demand in advance. When the demand from processor `A` comes, all the evaluated items of the list are shipped along the pipe. Alternatively, the pipe may implemented as a fixed-size buffer so that a fixed amount of evaluated items transfered each time. As such a function may affect the semantics, it must be checked first in a sequential setting to guarantee that it does not affect the semantics of the whole program before distributing it.

```
ys = pipe xs where
  pipe [] = []
  pipe (c:cs) = c `seq` (c : pipe cs)
```

The second is performance flexibility with respect to the user. One example of this aspect of performance is the ability to move functionality closer to

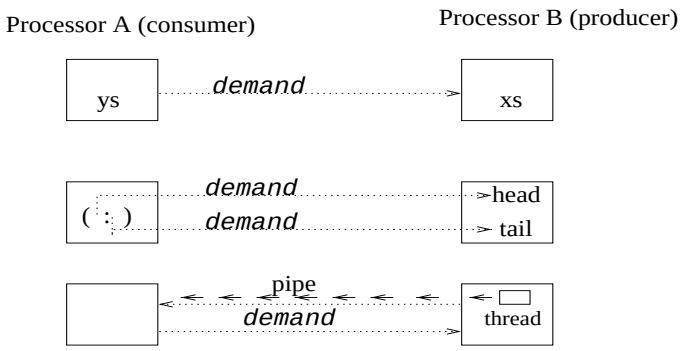



Figure 10.3: Pipes

the user; this can dramatically reduce perceived end-user latencies, which is a big issue in large-scale applications. Take as an example a program with a line-by-line interface. Typically there is an implicit line editing process (often supplied by the kernel). This handles the erase and kill processing as well as the character echoing. Distributing this program leads to the situation where the character editing has to do a complete round trip over the network, often giving uneven performance and is distracting to the user. Within this model, it is possible to migrate the line editing functionality to be close to the user, with the corresponding increase in predictability of performance of the echoing and line editing. The same principle can be applied to many validation of input tasks, or even large portions of applications.

The ability to "tune" the location of portions of the computation can be used to mitigate some of the end-user perceived effects of large-scale distribution. This ability to locate specific pieces of code near the user can dramatically hide the latencies that would otherwise be introduced. This increase in performance, as perceived by the user is not about minimisation of the computational time to complete a task but is about predictable and sufficient responsiveness to serve the user needs.

## 10.4 Mobility

Mobility is the ability of a language to change dynamically the location of execution of programs [38, 125]. This distribution model has two important char-

acteristics which make it mobile: first, it allows the shipping of code fragments with variable granularity and second, it is referentially transparent.

Cardelli [19] argues that computation is not just code but also the context of its execution. In Obliq [18], mobility is realised by moving closures containing the context of the computation. Our distributed system has the potential to support variable granularity of mobility, thus it enables the communication of any portion of the graph, together with any associated context of its execution. This is also realised by moving closures, since their use is common in functional language implementation. Heap objects are stored as closures containing references to code for executing them; these heap objects contain portions of the execution graph, which can either be data, code or computation. We have extended the concept of closure within a distributed context; marshalling suitable portions of graph in messages allows code and its associated context to be communicated. The implementation of this distributed system is based on the Brisk Machine [54], a variant of the `STG` machine [100]. As already described in Chapters 8 and 9, the Brisk Machine has been especially designed to be a flexible and dynamic model so that it allows the uniform and efficient implementation of multiple paradigms such as dynamic loading/linking and computational mobility. Functions are stored as ordinary heap nodes, which allows code to be communicated dynamically between machines and loaded into the current environment.

The preservation of referential transparency in a distributed environment has further implications, always orthogonal to the denotational semantics.

First, it provides a low latency distribution scheme, i.e. programs can start running without complete code; the missing code can be replaced by remote references as a form of dynamic binding. This derives from the fact that purely functional languages do not require a high degree of global consistency within the memory model, a property which is kept in a distributed environment since referential transparency is preserved.

Second, it results in a location independent distributed system. Through referential transparency, a distributed computation given the same inputs will always produce the same outputs, in the same order, independently of the pattern of distribution. The only difference will be the relative timings of which

these outputs appear. Computation can be moved on demand and the location of computation can change at run-time.

### 10.4.1 Global Computation

The global address space scheme which forms the operational basis of this distributed system offers an alternative approach to global computation. This is mainly due to the unique naming scheme it offers. As a natural consequence, it raises all the programming issues described by Cardelli [20]. We analyse them in the context of our model:

**Typing.** This mobile computation model is based on the non-strict purely functional paradigm and has the rich, polymorphic type system of Haskell. Type correctness is guaranteed on the one hand by static checking and on the other hand by the unique naming of objects in the presence of distribution.

**Security.** Cryptographic security is an important problem for distributed computation. Although sufficiently strong bulk encryption mechanisms already exist, the main challenge is to ensure the authentication of the end-points. Authentication of end-points needs to be resilient in the presence of masquerade attacks. In this distributed system, uniqueness of end-points is ensured by the the unique naming of objects in the global address space scheme, e.g. ASN object identifiers. These unique end-points can be used in conjunction with a trusted directory service or a hierarchical certification model like `X509`.

**Reliability.** This model has the underlying features that support reliability, e.g. the possibility of replication of values. In addition, the distributed garbage collection scheme we adopt provides support for isolating the non-available portions of memory, e.g. through temporary system unavailability.

**Modularity.** Modularity is needed to increase the potential for dynamic computation, i.e. to enable components to migrate at run-time. Wadler [131] argues that in functional languages, the explicit data flow together with the monadic approach provide the ultimate form of modularity, which is preserved in our distributed environment.

**Resource Management** The global address scheme provided in this distributed system enables the unique naming of objects and therefore provides a resource management scheme which is better than the usual name-based approach. Linking to facilities in this distributed system happens via a global identifier unique at any time and not just by name, which guarantees that the other end obtains a compatible version.

**Resilience** One further important programming issue in the context of global computation, beyond the ones mentioned by Cardelli, is resilience, which ensures trustability and repeatability in a distributed environment. In this distributed system, there is no possibility of introducing timing related dependencies when an application is distributed. This comes together with referential transparency which allows for equational reasoning. Whether the final distributed application performs its tasks at an acceptable rate is a different question, but at least the question of value correctness and ordering is dealt with.

## 10.5 Related Work

Erlang [135] is a concurrent functional language also used for distributed applications; although its distribution mechanism has been based on traditional concepts, its model does not need an interface description language and so it facilitates considerably the distribution process. Distributed Erlang is used in implementing large software projects within the Ericsson group.

Distributed Corba [113] interfaces traditional models into a distributed environment through the Interface Definition Language `IDL`. `IDL` is for Corba what `HTML` is for the Web and therefore it does not address any mobility issues.

Facile [40], [124] is an extension to `ML` with multiple processes and communication. Facile has been enriched to support weak mobility [66], in a distributed name space scheme. Executing units are implemented as processes which communicate via channels. Once a piece of code has been transfered, the channel is closed and the receiver may chose to send it to another node to be evaluated. Apart from being non-deterministic, Facile is different from the Brisk distributed model as it offers a distributed name space scheme which is in contrast to the global address space scheme provided by Brisk.

Java [41] supports weak mobility, i.e. the unit of distribution is fixed at design time. This is necessary because Java is implemented on top of traditional communications mechanisms which allow for non-deterministic effects and therefore fixing the unit and location of computation at design-time makes distribution semantically visible in denotational terms.

The Inferno [57] model has been designed as an operating system to be used in network environments. It provides an interface to resources scattered across the network and so distributed applications can be created. In Inferno, the interface to resources is fixed at design time, whereas the location of the code is statically assigned at run-time, i.e. it is the interface to resources that defines the granularity and the mobility.

Tarau and Dahl [123] presented a mobile system implemented on top of the BinProlog compiler, enriched with Linda [22] primitives and a continuation passing approach. The Linda model is highly non-deterministic; processes interact by selecting future work from a tuple space but where a particular computation occurs depends on accidental timings. The programmer has to ensure that such non-determinism does not give rise to unacceptable answers.

Obliq [18] supports strong mobility. It has both a graph metaphor and the ability to move demand. Obliq objects are local to the site of creation and therefore marshalling of computation needs to happen at the design stage.

The Mozart [129] distributed system implements Oz [115, 45] a concurrent, object-oriented language with dataflow synchronisation. Oz combines concurrent and distributed programming [44] with logical constraint-based inference, making it a unique choice for developing multi-agent systems. Agents represent a further step in expressiveness in concurrency than reactive systems. A reactive process reacts to events initiated by a user, whereas an agent acts itself as a user. Of course, agents suffer from the same non-determinism problems as reactive systems which arise as they communicate. Deterministic concurrency deals at present only with reactive systems and the distributed system that derives from it offers a trustable form of distribution across different processors due to the preservation of referential transparency.

Icarus [80] also supports strong mobility and has been designed as an extension to Occam [75]. Icarus preserves Occam's small granularity but removes its

fixed topology with mobile channels. Although it is a dynamic model, it does not provide variable granularity of mobility, once assigned it is fixed.

# Chapter 11

# The Distributed Brisk Machine

To support execution in a distributed setting and implement the operational model of the distributed system based on deterministic concurrency, as described in Chapter 10, the Brisk Machine has been added distribution extensions, in the form of built-in functions which convey information across machine boundaries and provide an interface to the global heap. This chapter describes an implementation built in the Brisk run-time system that emulates distributed execution on a single processor and demonstrates the validity of distribution based on deterministic concurrency. It also demonstrates that the novel feature of the Brisk machine and the Brisk compiler, namely to load functions as newly created nodes in the heap, enables computational mobility. This allows in a distributed setting heap fragments that represent data and code to be communicated dynamically between machines and loaded dynamically on another processor.

## 11.1 Introduction

In Chapter 10, it has been presented a model of distribution as a natural consequence of deterministic concurrency, a purely declarative form of concurrency that preserves referential transparency. This allows systems to be distributed without any change to their meaning, the only observable difference being the rate at which the system generates results. The operational model of this distributed system offers a transparent, global addressing scheme which ensures global knowledge of the current state of the system, where objects can be uniquely named. This offers a novel form of computational mobility where arbitrary portions of the program graph representing both code and data can be communicated between machines.

This chapter describes adaptations and extensions to the Brisk Machine that have been included to support such a form of distribution of computation and implement this operational model. These adaptations and extensions are built in the Brisk run-time system and presented in an implementation that consists of a collection of Brisk machines which exchange information via messages. and emulates distribution on a single processor. The global addressing scheme is realised by assigning each processor an index table to keep track of references from other machines.

Adding distribution extensions to the Brisk run-time system has been facilitated by the design of the Brisk machine. Distribution extensions that operationally convey information across machine boundaries and provide an interface to the global heap have been added in the form of built-in functions.

Furthermore, computational mobility is facilitated by the uniform representation of expressions in the Brisk machine, including functions as heap nodes. This provides support heap fragments that represent data and code to be communicated dynamically between machines or loaded dynamically into the running environment.

## 11.2 Extensions

A simulator for distributed execution is presented which implements a global distributed heap. It consists of a collection of machines, which communicate by

exchanging messages. Each machine contains heap allocated objects that form part of the overall graph as specified by annotations to the program.

The main extensions to the brisk run-time system in order to support distributed execution are:

- a new run-time module, the Distributor Module, to handle distribution;
- adding pointers to procedures as registers in the machine state for allowing execution code to communicate indirectly with the run-time system, see also Figure 11.1;
- special built in functions to handle distribution details, e.g. executing the code of messages.

The run-time system is now:

- The Heap Module;
- The Module Manager (Loader) Module;
- The Scheduler Module;
- The Distributor Module;
- The Run Module.

## 11.3 The Run Module

For the purposes of distribution, the structure of registers that hold parts of the machine state has been extended to include also registers that hold pointers to special procedures e.g. for sending a message or creating a new machine, see also Figure 11.1. This technique has been borrowed from the Java implementation. This has been necessary, since allowing execution code to access items from the run-time system only through macros resulted to be too restrictive in a distributed setting. Furthermore, this technique has one more advantage. it allows the execution code to call these procedures without altering the machine state, i.e. no need to save the machine state in the registers and restore it later.

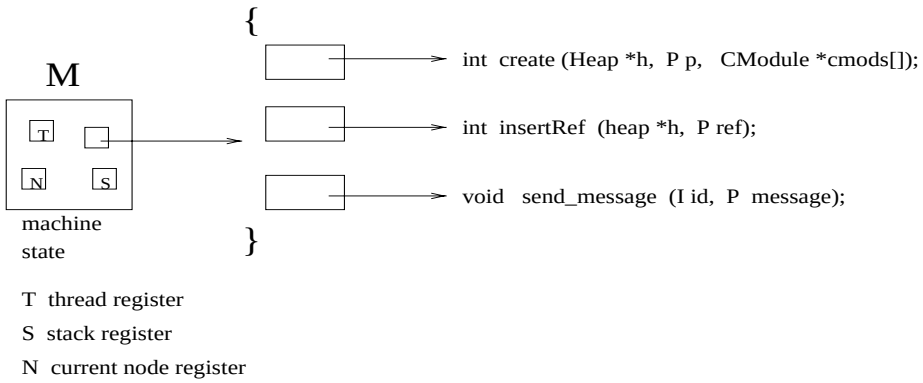



Figure 11.1: Registers that hold Pointers to Functions

```
#define BRISK_REGISTERS
  P N;      /* current node register */      \
  P S;      /* stack register */             \
  P T;      /* thread register */            \
  P H;      /* free heap memory register */  \
  P L;      /* heap memory limit register */ \
  I C;      /* condition register */         \
  I ID;     /* machine identifier */         \
  heap *h;   /* the heap structure */                 \
  /* create a new machine */
  int (*create)(heap *h, I id, P node, CModule *cmods[]);\
  /* send a message to a machine */
  void (*send_message)(I id, P message);               \
  /* insert a node in the index table */
  int (*insertRef)(heap *h, P ref);                    \
  ...
typedef struct mc
   BRISK_REGISTERS;
 mc;
```

## 11.4 The Distributor Module

The run-time system has been augmented with a special module, the distributor module, especially designed to provide support for distributed execution and to handle distribution issues. More precisely, this module offers the following functionality:

- it deals with incoming messages;
- it responds to requests for creating new machines;
- it responds to requests for sending messages;
- it swaps machines when the control of execution is transfered to some other machine;
- it calls the scheduler for each machine.

Distributed execution is handled by an array of machines together with the distributor node. The array of machines contains the available machines of the system, where each machine may hold parts of the program graph.

The distributor node is a heap node stored in the root node. Its main role is to deal with messages between machines and to transfer execution between them, as specified by program annotations. It contains 4 fields, see also Figure 11.2:

- an info node;
- a status that indicates whether a particular machine is idle or not;
- a pointer to the index table;
- a pointer to the queue of incoming messages;

The index table records the pieces of graph that are referred to from outside of a machine and so, it implements the global address space of this distributed system. It offers global knowledge of the inter-processor aspects of the system and acts as a mechanism that decouples the local heap from the global distributed heap. The index table, see also Figure 11.2 contains:

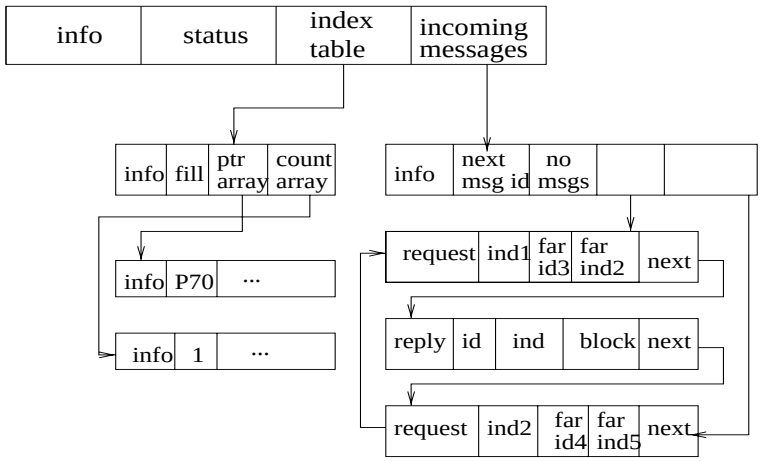



Figure 11.2: The Distributor Node

- an info node;
- an integer to indicate the number of entries already filled in the array of pointers/counters;
- an array of pointers, implemented as a flexible node, which records the required nodes from other machines;
- an array of counters, implemented as a flexible array for raw data, which matches the array of pointers and indicates how many references there exist for every node in it.

The queue of messages has an info node, the identifier of the next message, the total number of messages and pointers to the first and the last item of a circular queue of messages.

When a distributed program runs, control is transfered to the distributor. This performs repeatedly the following steps for each machine in a round-robin fashion until there are no messages in transit and no threads left in any of the machines:

- if the status of the machine is idle and there are no messages in the message queue, control of execution is transferred to the next machine;

- if there are messages in the message queue, it dequeues them and sets the machine active if it is not;
- for each message, a new thread is created and enqueued to the scheduler and the message node becomes the current node of each such thread;
- the scheduler is called for a quantum to execute threads in the current machine.[1]

```
while (1) {
 for (i = 0; i < nmachines; i++) {
   distributor = getPtrWord(machines[i]->roots,distribRoot);
   mes_queue = getPtrWord(distributor,5);
   n = getNumMessages(mes_queue);
   f ((isIdle[i]==0) && (n == 0)) continue;
   if ((isIdle[i] == 0) && (n != 0)) {
       setProcessorActive (distributor);
       isIdle[i] = 1;
   }
   for (k = 1; k <= n; k++) {
       message = (P)dequeueMessage(mes_queue);
       thread = newThread (machines[i]);
       putPtrWord(thread, 1, 0);
       putPtrWord(thread, 2, message);
       putPtrWord(thread, 3, thread);
       putPtrWord(thread, 4, thread);
/*   spawn the new thread */
       spawn(machines[i], thread);
   }
   result = schedule(machines[i]);
  ...
}
```

[1] Macro `getPtrWord(p,n)` returns the node which resides at the `n`th word of the node `p`, whereas `putPtrWord(p,i,p1)` puts a pointer to the node `p1` in the `i`th word of node `p`.

In a distributed setting, the scheduler runs for a quantum,[2] calling repeatedly the execution code of each runnable thread until either all threads finish, or the quantum expires or the execution code requires something special to happen. In all these cases, a condition is returned as an integer to the scheduler which is passed also to the distributor. This integer condition may either specify the condition of a thread (blocked, waiting, creating a new thread etc), in which case the scheduler deals with it by invoking the appropriate function as described in Chapter 9, or the condition of a machine, e.g, a machine is going idle, in which case control is given to the distributor. As many issues concerning distribution have been dealt by adding pointers to special procedures as registers at the machine state, there are relatively few conditions concerning machines.

## 11.5 Extensions to Support Distributed Execution

### 11.5.1 Mapping Graph Fragments to Machines

Distributed execution is specified by annotations that have been inserted to specific points of the program graph. These annotations indicate which parts are going to be executed remotely. Initially, a machine is assigned to be the parent and the whole program is loaded to it. As simple example of such an annotated program written in `BKL` is as follows:

```
main = putInt fortytwo;
fortytwo = plus fartwenty twentytwo;
fartwenty = remote 2 twenty;
twenty = MkI 20;
twentytwo = MkI 22;
```

The built-in *remote* introduces distributed computation and specifies which graph fragments belong to other machines and are going to be executed remotely. The general form of annotations that specify distributed execution is:

[2] In a sequential setting the scheduler runs infinitely until all threads have finished, whereas in a distributed setting the scheduler runs for a quantum and then transfers control to the distributor.

cn $\mapsto <$ *remote*, *id*, *graph_piece* $>$

remote $\mapsto <\ldots, C_{remote}, far>$

The code `C_remote` is defined in a special core module (`PreludeDist.hs`) that handles distribution issues:

```
remote = primCFremote 0 1 far;
{-#include
static void remote (MC *M)
{
P far,p,node,distributor;
I index;

far = GLB((M->N)[0],1);
node = ARG(1);
id = ARG(2);
index =  (M->create)((M->h), id, node, cmods);
CHK(3);
p = NODE3(far,(W)id,(W)index);
ENTER(p);
}
#-}
```

When such a node[3] is encountered, its code performs the following tasks:

- it initialises a new empty machine with identifier `id` together with a new loader, scheduler and distributor node and records the new machine in the array of machines. This is achieved through the `int create (id, node,`

[3] A number of macros have been defined. Macro `GLB(p, i)` returns the `i`th reference of a given node `p` that resides in its info node. Macro `ARG(i)` returns the `i`th argument of given node. Macro `CHK(n)` checks for a given amount of heap space. Macro `NODE3(a,b,c)` allocates a node of size 3 on the heap, makes `p` point to it, and fills it in with the given values `a`,`b`, `c`. As an alternative to `CHK(n)/NODE3` pair of macros, the `NEW(n)/FILL(n)` pairs can be used for allocating nodes more than size 5. Macro `Enter(p)` makes node `p` the current node. Also `primCFname unboxed boxed references` represents a built-in function implemented by a `C` function with the given name, that expects a given number of raw and boxed arguments. The listed global references are available for use by the `C` function.

`cmods)` function, defined in the distributor module, a pointer of which has been included as a register to the machine state. [4]

- the node that represents the graph fragment that needs to be communicated is unloaded from the parent machine and loaded to the new machine with identifier `id`. After loading the graph fragment in the new machine, an entry to the node that corresponds to this graph fragment is recorded into the index table and the index table offset is returned back to the parent machine. Loading and unloading graph fragments from a heap is explained analytically in the next Section 11.5.2.

- a node `far id index` is created in the parent machine to act as a substitute for the remote node.

### 11.5.2 Unloading and Loading of Heap Fragments

**Unloading of Heap Fragments** This paragraph describes the procedure of creating a block of raw data which represents a graph fragment that needs to be communicated. Again, this is facilitated by the uniform representation of nodes in the Brisk machine, where functions also are stored as heap nodes, while providing layout information for call nodes.

If the node which represents the piece of graph that needs to communicated is a global declarative entity, both its `BID` together with the `BID` the module it belongs are extracted from the loader. The `BID` of the module is used to load this module in the new machine. The `BID` of the entity in the new machine is used to find its corresponding heap node in the new heap. This node is entered into the index table.

```
if (hasBid (h,p)) {
   p_bid = find_bid(h, p);
   m_bid = findbidBModule(h,p_bid);
   loadB (new_h,m_bid);
```

[4] Note the usage of `create` function, a pointer of which has been included as a register to the machine state, from the execution code. As it has been included as a register to the machine state, it is used as `index = (M->create)((M->h),id,node,cmods)` from the execution code, whereas as `index = create(h, id, node, cmods)` from the run-time system.

```
    node = findEntity (new_h, p_bid);
    offset = insertRef(new_h, node);
}
```

If the node that represents the piece of graph that needs to be communicated is not globally known, i.e. it has not a `BID`, then a block is created to hold the required piece of graph as raw data. The creation of block happens in three passes:

In the first pass, the node that needs to be transfered is scanned word by word to find out what other non-global pointers it references. These are recorded in a flex node called pointer table.

```
p = (P) node;
ip = *p;
for (i = 0; i < getISize (ip);) {
   p = ((P)node) + i;
   /* if p is global, ignore it */
   if ((hasBid(h, *p)) {
      /* move to the next pointer word */
      /* if p is an info node */
      if (nrefs == 0) {
      /* skip the pointer accordingly */
          i = i + 1 + getIShape ((P)(*p));
      /* extract the references */
          nrefs = getISize ((P)(*p))-1-getIShape ((P)(*p));
      } else {
      /* if p is not an info node */
      /* scan the next position */
          i++;
          nrefs--;
      }
         continue;
   }
   /* if p is not global */
   else {
```

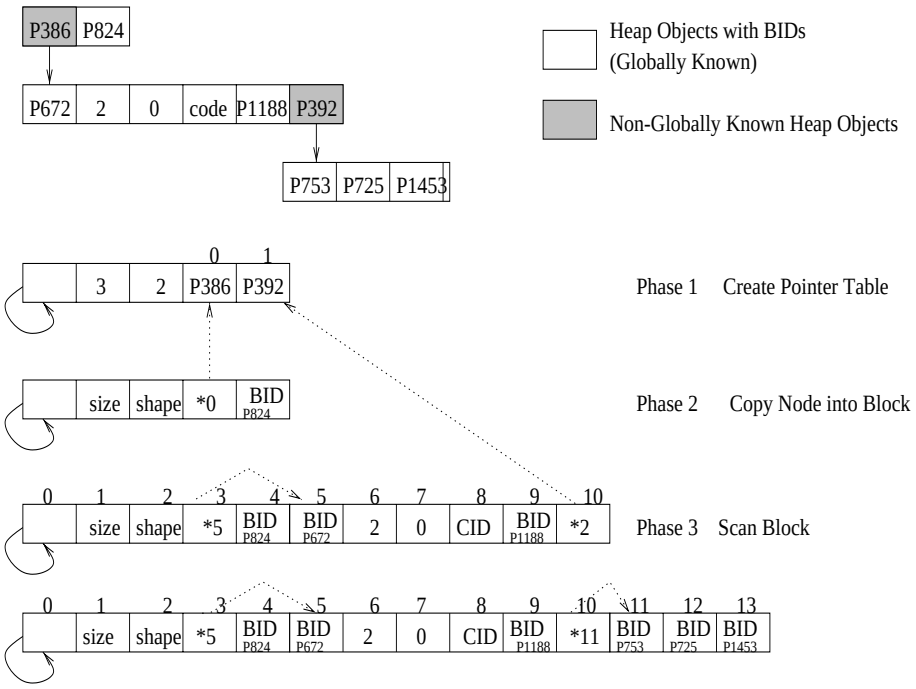



Figure 11.3: Creating a Message from a Non-Global Entity

```
    /* insert a pointer in the pointer table */
       table = checkFlex (h, table, 1);
       pushFlex (table, ((P)*p));
       /* move to the next pointer word */
       ...
    }
}
```

Each entry in the pointer table is also scanned to find what other non-global pointers it references. Pointers to these non-global entities are also recorded into the pointer table and scanned until all relevant non-globally known pointers are loaded.

In the second pass, a block is created as a self-descriptive node to hold the heap fragment as a collection of raw data. Copying a node that needs to be transfered into the block involves scanning this node word by word and inserting a corresponding integer for each word in the block. During scanning:

- if a word is a reference (pointer), it is checked whether its is a global declarative entity or not. If it is global, its `BID` is recorded in the block. If it is not global, then there is already an entry for this reference in the pointer table from the previous phase. In this case, its relative address from the pointer table is recorded in the block, see also Figure 11.3. `BID` are distinguished from relative addresses as the latter have their top bit set. In both cases, if the reference is the info pointer of the node, the number of raw words and references of the node are extracted.

```
p = (P) node;
ip = *p;
for (i = 0; i < getISize (ip);) {
  p = ((P)node) + i;
  q = ((P)block) + wrapper + i;
  /* if p is global  */
  if (hasBid(h, *p)) {
    /* find its BID */
    bid = find_bid(h, *p);
    /* insert it into the block */
    (I)*q = bid;
    /* If the node is an info node */
    if (nrefs == 0) {
       /* copy its raw words in the block */
       for (k = 0; k < getIShape (ip); k++) {
           a = (I)*(node+i+1+k);
           q = ((P)block) + wrapper + i + 1 + k;
           /* and the code cid, if any */
           if (k == 2) {
              (I)*q = (cid) find_cid(h,(C)a);
           } else {
              (I)*q = a; }
       }
       /* extract the number of references and
          skip to the next position. */
```

```
            i = i + 1 + getIShape (ip);
            nrefs = getISize (ip) - 1 - getIShape (ip);
          } else {
            /* if the node is not an info node,
               move on to the next position */
            i++;
            nrefs--;
            }
          continue;
        }
        else {
        /* if p is not global,
           find its index in the pointer table */
            n = (I)indexTable(h, (*p), table);
            /* insert it as a relative address in block */
            (int)*q = (I)SET(TOPBIT1, n);
            ...
        }
      }
```

- if a word is an integer, it is checked whether it is a code entry point and if so, it is replaced by its `CID` identifier, otherwise the integer is copied straight into the block.

In the third pass, the block is scanned again word by word. Now, the block contains only integers, either `BID`s for the global declarative entities or relative addresses to the pointer table. The latter have their top bit set. During scanning, each word is checked whether it is a relative address in the pointer table or not.

- If a word is a relative address in the pointer table, the corresponding node is extracted from the pointer table and copied into the block by repeating pass two for this node. The relative address in the pointer table becomes now a relative address in the block, pointing to the position in the block the new node has been added, see also Figure 11.3.

```
for (i = wrapper; i < getISize(block); i++) {
  p = ((P) block) + i;
  /* if p is a relative address in the pointer table */
  if ((ISSET(TOPBIT1, *p))) {
    /* find the index in the pointer table */
    n = (I)CLR(TOPBIT1, *p);
    /* get the node from the pointer table */
    node = indexFlex (table, n);
    ptr = getRawWord (block, 3);
    /* add the node in the block */
    block = (P)copyNode(h,block,node,table,wrapper+ptr);
    /* make p a relative address in the block */
    (I) *p = (I) SET(TOPBIT1, wrapper+ptr);
   }
}
```

- If the word is a `BID`, it is left as it is.

The above procedure continues until all the words in the block have been scanned, all relevant nodes copied in the block and all pointers translated from relative addresses in the pointer table to relative addresses in the block. The block is now a collection of raw data that can be loaded to the other end.

**Loading of Graph Fragments** After the block of raw data that represents the graph fragment is created in the parent machine, it is copied in a suitable node in the target machine. Similarly to the dynamic loading of modules, dynamic loading of heap fragments in the new machine happens in two passes.

In the first pass the block of raw data that represents the graph fragment is scanned word by word to find out what other unloaded modules it references. These modules are also read into the block and scanned, until all relevant module templates are loaded. During scanning, if a reference is a relative address in the block, it is further checked to see if it is the first word of a node, i.e. an info node. If so, the raw words are skipped by advancing the pointer accordingly and the number of references is extracted. If not, the number of references is decremented and the pointer is advanced to the next position.

If a reference is an entity identifier, the `BID` of its module is found using the loader node. If the module is neither loaded nor in the block, the module is added at the end of the block and a new block is returned. Again, if a reference is the first word of the node the number of references is decremented and the is pointer advanced to the next position. The scanning continues until all the words in the block have been scanned and all the relevant module templates added in the block. The resulting block consists of a collection of raw data which contains the graph fragment together with any modules it might reference.

The second pass involves linking and installing the modules into the module table. During linking, the block is scanned again and every reference is translated into a pointer in the heap of the new machine. If a reference is a relative address, it is translated into a heap pointer. If it is a global declarative entity, its module is found either already installed or in the block of templates, and the reference is translated into a heap pointer. Also the type field of each node is checked to see if it has an indication of a `C` code pointer. If so, the code identifier `CID` is replaced by a pointer to its code entry.

After translating every reference into a heap pointer, the first part of the resulting block corresponds to the communicated graph fragment and the rest to any other modules it might reference. A pointer to the communicated graph fragment is entered in the index table of the new machine and the index table slot is returned to the parent machine. Finally, the rest of the block, i.e. the module templates are installed in the `B` module table, by setting in each `B` module entry a pointer to a flexible node that points to the heap pointers of those declarative identifiers that are exported.

### 11.5.3 Remote References

After the initialisation phase, all graph fragments that are supposed to be executed remotely have been moved from the parent machine and loaded in the appropriate machines. In the parent machine, they are now remote references to substitute them.

A remote reference has the form `far id index`. It contains an info pointer and two raw integers. The first one indicates which machine is the owner of the graph fragment it substitutes and the second specifies the offset in an index

table in the remote machine which points to this graph fragment.

cn $\mapsto < far, far\_id, far\_index >$

far $\mapsto < \ldots, C_{far}, request, wait >$

The code `C_far` defined in `PreludeDist.hs` is:

```
far = primCFfar 2 0 restore_thread request wait;
{-#include
static void far (MC *M)
{
   P distributor, request;
   P table, message, wait;
   I my_id, offset, far_id, far_index;

   request = GLB((M->N)[0],1);
   wait = GLB((M->N)[0],2);
   (M->N)[0] = wait;
   my_id = (M->ID);
   far_id = (I)ARG(1);
   far_index = (I)ARG(2);
   distributor = getPtrWord((M ->h)->roots, distribRoot);
   table = getPtrWord (distributor, 2);
   offset = (M->insertRef)((M->h), (M->T));
   message=NEW(5);
   FILL5(message, request, (W)far_index,
        (W)my_id), (W)offset, message);
   (M -> send_message) (far_id, message);
}
#-}
```

When such a remote reference becomes the current node and its code is executed, the following tasks are performed:

- the current node `far id index` mutates into a node `wait id index`. The code of the `wait` function calls a suitable macro from the run-time system which causes the condition of the current thread to be updated to a

special value and control is given back to the distributor which calls the scheduler. The scheduler increases the number of waiting threads but does not enqueue the thread back to the queue of runnable threads;

- the current thread is stored in the index table of the parent machine and its offset in the index table is returned. This offset is communicated into the message and will act as the target for the reply. When the reply message will arrive and its code executed, it will restore the suspended thread, communicate with the scheduler to decrease the number of waiting threads and enqueue the thread back to the queue of runnable threads;

- a request message is constructed. This takes as arguments the index table offset in the remote machine which points to the required piece of graph, the identifier of the current machine and the index table offset which holds the suspended thread;

- the request message is sent to the machine with identifier `id` via the `send_message (id, message)` function. This takes as arguments the remote machine identifier and the message itself and causes the message to be added to the queue of messages of the remote machine. This function has been defined in the distributor module and a pointer to it included as a register in the machine state.

### 11.5.4 Messages

Messages transmit the control of execution remotely by carrying demand across machine boundaries, causing the evaluation to happen in a different machine. Incoming messages are queued in the message queue that resides in the distributor node. The queue of messages has no information about the execution model of each message. Instead, according to the active messages approach [32], each message contains its own execution code which is executed on arrival at the target machine. There are two main kinds of messages: *request* and *reply* ones.

**Request Messages** A request message mediates the flow of demand and therefore control of execution to the owner of the requested piece of graph. Such a message has 5 fields:

- an info pointer;
- the identifier of the remote machine that holds the requested piece of graph;
- the index table offset of the remote machine which points to the requested piece of graph;
- the index table offset of the requester machine which points to the suspended thread and acts as a target for the reply.
- a field used for chaining when the message is enqueued at the message queue.

A request message has the form:

cn $\mapsto <$ *request*, *far_index*, *req_id*, *req_index*, *request* $>$

request $\mapsto <$ ..., $C_{request}$, *fr*, *send_reply* $>$

where `far_index` is the index table offset of the remote machine that points to the required piece of graph, `req_id`, `req_index` are respectively the identifier of the requester machine and its index table offset which points to the suspended thread.

The code `C_request` is also defined in `PreludeDist.hs`:

```
request = primCFrequest 3 1 evaluate_send_reply;
{-#include static void request(MC *M)
{
  P distributor, table, refarray, countarray;
  P node, nd, evaluate_send_reply;
  I req_id, req_index, far_index, fill;

  evaluate_send_reply = GLB((M->N)[0],1);
  far_index = (I)ARG(1);
  req_id = (I)ARG(2);
  req_index = (I)ARG(3);
  distributor = getPtrWord((M->h)->roots, distribRoot);
```

```
  table = getPtrWord (distributor, 2);
  refarray = getRefArray(table);
  countarray = getCountArray(table);
  node = refarray[far_index];
  (M->decCounter)((M->h), far_index);
  nd = NEW(4);
  FILL4(nd,evaluate_send_reply,(W)req_id,(W)req_index,node);
  ENTER(nd);
}
#-}
```

When the code of a request message is executed in the target machine, it uses the `far_index` offset of its index table to access the requested node. An entry to this node is already into the index table from the initialisation phase. Executing the code of a request message causes:

- the requested node to be accessed in the heap through its index table offset in the request message;
- the reference counter of the requested node to be decremented in the index table;
- a new node `evaluate_send_reply` to be created in the heap; its purpose is to evaluate the requested piece of graph and to construct a reply message which will be send to the requester.

The `evaluate_send_reply` node has the form:

cn $\mapsto <$ *evaluate_send_reply*, *req_id*, *req_index*, *node* $>$

evaluate_send_reply $\mapsto <$ ..., $C_{strict001}$, $fr$, *send_reply* $>$

where `fr` is a stack frame constructor. The code of `strict001` is as follows:

```
{-#include
static void strict001 (MC *M)
{
   P str, f1;
```

```
   CHK(5);
   str = GLB((M->N)[0],1);
   f1 = GLB((M->N)[0],2);
   (M->S) = NODE5(str, (W)0x4, (M->S), f1, (M->N));
   (M->N) = ARG(3);
}
#-}
```

The built-in `strict001` indicates that only the last of the three arguments of the `evaluate_send_reply` function, i.e. `node` requires evaluation. The code of the built-in `strict001` creates a stack frame corresponding to the suspended call and makes the third argument, i.e. `node` the current node. When the evaluation of the node finishes, the suspended call is popped off the stack. Its info node is updated in place to point to the continuation function `send_reply`, the evaluated argument (`node`) is updated in place to point to its evaluated form `evaluated_node` and the expression is made the current node.

cn $\mapsto <$ *send_reply*, *req_id*, *req_index*, *evaluated_node* $>$

send_reply $\mapsto <$ ..., $C_{send\_reply}$, *reply* $>$

The code `C_send_reply` is as follows:

```
send_reply = primCFsend_reply 2 1 reply;
{-#include
#include <stdio.h>
static void send_reply (MC *M)
{
   I req_id, req_index, format;
   P evaluated_node, block, reply, message;

   reply = GLB(NR[0],1);
   req_id = (I)ARG(1);
   req_index = (I)ARG(2);
   evaluated_node =  ARG(3);
   block = (M -> createBlock)((M->h), evaluated_node);
```

```
    message = NEW(5);
    FILL4(message, reply, (W)req_index, block, message);
    (M -> send_message) (req_id, message);
    DIE();
}
#-}
```

The code of the `send_reply` function performs the following operations:

- it creates a block of raw data as described in section 11.5.2;
- it creates a reply message; a reply message consists of an info node, the index table offset at the target machine that points to the suspended thread, the block of raw data and a pointer that is used for chaining when the message is enqueued at the message queue;
- it sends the message to the appropriate machine via the `send_message` function; this takes as arguments the remote machine identifier and the message itself and when executed causes the message to be added to the queue of messages of the remote machine;
- it discards the thread created by the request message calling a special macro, which causes the machine to save its registers and to return control to the distributor which calls the scheduler. The scheduler does not enqueue the thread back in the runnable queue and sets the machine idle if there is no other activity associated with it.

**Reply Messages** Reply messages transfer values or graph fragments across machine boundaries in response to request messages. A reply message has 4 fields:

- an info pointer;
- the index table offset by which the suspended computation can be accessed in the target machine;
- a block which holds the reply in the form of raw data.

- a `next` field used for chaining when the message is enqueued in the message queue.

  cn $\mapsto < reply, req\_index, block, next >$

  request $\mapsto < \ldots, C_{reply}, \ldots >$

  The code `C_reply` is also defined in `PreludeDist.hs`:

```
reply = primCFreply 1 2;
{-#include
static void reply (MC *M)
{
   I req_index, format;
   P block;

   req_index = (I) ARG(1);
   block = (P) ARG(2);
   new_block = (M->copyBlock) ((M->h), block);
   node = (M -> loadFragment) ((M->h), new_block);
   old_thread = (M->getRef)((M->h), req_index);
   (M->N) = node;
   (M->T)[2] = node;
   (M->T)[3] = old_thread[3];
   (M->T)[4] = old_thread[4];
   (M->S) = old_thread[3];
   DIE();
}
#-}
```

When the code of a reply message is executed, it code performs the following:

- it loads the block of raw data in the heap and returns a pointer to the node that points to this fragment, as described in Section 11.5.2;
- it restores the suspended thread, accessing it from the index table using the index table offset from the reply message;

- it makes the current node register of the suspended thread the node that points to the graph fragment that has been loaded;
- it makes the suspended thread runnable. Making it runnable involves calling the macro `UNWAIT(p)` from the run-time system. This returns a special condition, which causes the machine registers to be saved and control is returned to the distributor. The distributor calls the scheduler, which decreases the number of waiting threads and enqueues the suspended thread back in the queue of runnable threads;
- it makes the new node the current node register;
- it discards the thread created from the reply message.

## 11.6 Execution in a Distributed Setting

### 11.6.1 Initialisation

Distributed execution starts by initialising a parent machine, i.e. a new heap together with a loader, a scheduler node and a distributor node. The whole distributed program is loaded to the parent machine as described in Chapter 9. A new thread is created which is enqueued to the scheduler, its current node register points to the expression `runIO main World` and its stack register to a a special `finish` node, the code of which causes the result to be printed. The parent machine is recorded to the array of machines and its index becomes the machine identifier stored in its heap structure.

As in a sequential setting, evaluation starts in the parent machine by evaluating the code of the right hand side of the definition of `main`. A new thread `runIO main World` is created to evaluate `main`, which is enqueued to the scheduler and the distributor takes over. When `remote` annotations are encountered, the respective machine is initialised and the corresponding piece of graph is shipped to it. Later we plan to skip the initialisation phase so that during compilation, the appropriate piece of graph is initialised at the appropriate machine.

### 11.6.2 Distributed Execution

When annotations that specify remote execution are encountered, the specified piece of graph is assigned to the appropriate remote machine, described in Section 11.5.1. An index table entry is inserted in each machine for each remotely allocated piece of graph representing the migrated subexpression.

The distributor checks whether there are any incoming messages for the current machine. If so, it creates one thread for each message and enqueues them to the scheduler. Then, control of execution is passed to the scheduler. This runs for a quantum executing the code of each runnable thread in turn for one time-slice until either all threads finish, or the quantum expires. The code of each runnable thread loads the machine registers and repeatedly calls the execution code of the current node register. If the execution code of the current node register requires something special to happen, it either calls one of the procedures, pointers of which have been included as registers in the machine state (e.g. to create a new machine or to send a message) or it returns a special condition to the distributor to satisfy the request (e.g. to set the machine idle or to create a new thread). In the latter case, control of execution is transfered to the distributor which satisfies requests that concern another machine or calls the scheduler to satisfy requests for threads.

When the time-slice of a thread finishes, an integer condition is returned to the scheduler. The integer condition indicates either:

- what the scheduler should do with that thread;
- whether control should be returned to the distributor;

### 11.6.3 Termination

Distributed execution terminates when there are no messages in transit and no threads left in any of the machines. When the current node of the last thread, in any machine other than the parent, reaches weak head normal form, its stack node has a special code which arranges the machine to become idle. When the current node of the last thread of any other machine reaches weak head normal form, its stack node has a special code which arranges this particular machine to become idle.

# Part V

# Conclusions

# Chapter 12

# Conclusions and Further Work

## 12.1 The Brisk Machine

This thesis has presented the Brisk machine, especially designed so that it provides a flexible and thus an extensible machine to allow for general experimentation with different execution models. To achieve this, the compiler's intermediate language `BKL` (Brisk Kernel Language) is lower level than is usual. This simplifies the run-time system and makes the execution of programs match closely the graph reduction model. Several extensions concerning optimisations, concurrency, distribution and logic programming have been added without any interference between them. The Brisk machine is also a dynamic model, since expressions, including functions, have a direct and uniform representation in the heap, stored as heap nodes. This helps support both dynamic loading and computational mobility, where code can either be communicated dynamically between machines or loaded dynamically into the running environment. Run-time system techniques have been presented, as incorporated in the Brisk compiler which allow for dynamic loading and computational mobility in Haskell compilers. Several of the above extensions have been added:

**Dynamic loading** of newly compiled modules is supported by the ability to load functions as newly created nodes in the heap and to mix compiled and

interpreted bytecode on a function-by-function basis.

**Distribution and Computational mobility** in a distributed setting has been facilitated by the direct and uniform representation of all expressions, including functions, in the heap. This allows functions to be shipped between processors. The deterministic concurrency approach allows communication between processors to be implemented in such a way that it is equivalent to a single distributed heap; this is eased by the ability to insert built-in functions to hide the communication details. A distributed program is thus kept equivalent to its non-distributed counterpart, which helps to make distribution orthogonal to other issues.

**Portability** in a distributed environment is supported by a **bytecode interpreter for Haskell** which has been incorporated into Brisk's run-time system. The main idea behind this, is that Brisk provides apart from the intermediate language `BKL`, bytecode instructions that form a conventional abstract machine similar to the `G`-machine. This is a novel approach in compilation of functional languages since so far Haskell compilers compile via either machine instructions or an intermediate language. The Brisk compiler compiles via both. A similar approach has been adopted in the Glasgow Haskell Compiler [95] but the Brisk machine has been the first working implementation of this combination for a non-strict functional language.

**Debugging tools** are also supported by the uniform representation of nodes. This, as explained by Penney [93], is facilitated by the ease with which the layout of nodes can be changed dynamically and the close relationship of the run-time system to simple graph reduction, without complex stacks and registers.

**Logic programming extensions** in the form of Escher programming language [76] have also been added as an extension to the Brisk machine. This involved adding run-time variables (in the logic programming sense), changing the details of execution order and changing the approach to such things as strictness, largely by replacing built-in functions by alternatives.

### 12.1.1 Further Work

More work is needed to add all the optimisations to the Brisk machine and investigate its efficiency, particularly in comparison to the `STG` Machine. It would be interesting to compare the Brisk machine approach to partial applications with the `STG`-machine approach to run-time checks.

## 12.2 Deterministic Concurrency

Another major contribution of this thesis is that it dealt with preserving referential transparency in the presence of concurrency. Certainly, the two do not cohabit well, so deterministic concurrency offers an acceptable compromise. The deterministic concurrency idea has been initiated by Holyer and Carter [23, 51] by providing distinct threads that do not communicate. Together with careful design this has been proved expressive enough to provide a single-user multitasking working environment including shells, file managers and other operating system features.

In this thesis, the model of deterministic concurrency which has initially been built on top of Haskell 1.2, has been adapted to the monadic `I/O` model. Special concurrency primitives have been provided to spawn independent threads of execution. This has been achieved by *state splitting*, a technique which assigns each new thread part of the state, a *substate*, to act upon. In this model, distinct concurrent threads are restricted to access disjoint substates. This is in contrast with the approach taken in Concurrent Haskell where the `forkIO` primitive allows parent and child threads to mutate the same state which immediately introduces non-determinism, e.g. two threads can access the same file simultaneously.

As a result, the Brisk monadic framework has been presented with incorporates deterministic concurrency in Haskell 1.4 on top of the monadic `I/O` framework [106]. This provides both facilities for creating independent substates (State splitting) and to allow actions on them. Such an extended monadic framework offers some additional advantages:

1. It provides a modularised approach to state types, where each state type and primitive operations can be defined in its own module.

2. It improves some existing language features by making them deterministic such as reading characters lazily from a file without unexpected effects. This can be used to write a Haskell definition of the `hGetContents` facility available from current Haskell systems.

3. It provides a lazy version of the standard Haskell function `writeFile` for writing characters lazily to a file.

The Brisk monadic framework has been prototyped in an older version of the compiler, as the current one has not yet been added a type system. Nevertheless, this first attempt proved its feasibility.

This thesis has not only attempted to adapt the above mentioned deterministic concurrency model so that it fits within the monadic `I/O` in Haskell but also to increase its expressiveness by providing a deterministic form of communications. A freer style of communication has been adopted where arbitrary networks of components, communicating via deterministic time-merged channels, are set up. This time-merging has been based on a hierarchical notion of time which imposes an ordering of communication events within the threads[56]. The use of timestamps on messages together with synchronisation messages ensures, in a transparent way, that all merging of message streams in a program is deterministic, without any unnecessary delays. The timestamping idea can have an important impact in `GUI` design as it forces deterministic behaviour whereas it retains modularity of `GUI` components in an object-oriented style. It could be used to provide deterministic libraries for existing `GUI`s such as e.g. Haggis or for Java. The timestamping approach also can be used as a mechanism for incorporating multi-user facilities in a way which reduces their inherent non-determinism to a minimum.

### 12.2.1 Limitations of Deterministic Concurrency

The initial model of deterministic concurrency assumed a number of distinct threads that do not communicate [23, 51]. This has been proven expressive enough to provide a single-user multi-tasking working environment including shells, file managers and other single-user operating system features. Of course, this form of concurrency is a compromise and requires very careful design in or-

der to provide the above facilities in a deterministic way. In addition, this form of concurrency is very limited as it cannot be used as a substrate to build multi-user facilities and operating systems, since they are inherently non-deterministic. For example, when multiple users attempt to update a database, explicit timing information is required to preserve determinism, at least to some extent. This form of concurrency cannot provide a solution to the dining philosophers problem or the readers and writers problem as they encapsulate concurrency aspects that characterise multi-user aspects of concurrency.

This thesis has attempted not only to adapt the above mentioned deterministic concurrency model in the monadic `I/O` model of Haskell but also to increase its expressiveness by providing a deterministic form of communications. The timestamping approach that has been introduced, can offer graphical user interfaces in an object-oriented style where good modularity is an advantage without at the same time employing a non-deterministic select/merge primitive. It also can contribute in the design of multi-user functional operating systems, where non-determinism can be considerably restricted. Of course the timestamping approach restricts efficiency, as a message is not allowed to proceed until it is determined that no earlier message can arrive at a merge point, but this form of timestamping is addressed mainly to provide graphical user interfaces where the only requirement is to handle smoothly user event such as mouse events and thus, efficiency at the speed of mouse moves is acceptable.

### 12.2.2 Further Work

The timestamping solution as presented in this thesis is unoptimised. More work is needed to investigate solutions that eliminate the number of synchronisation messages generated by the system.

Also, in this thesis we considered only static networks of processes. The timestamping approach needs to be extended to allow for dynamic reconfigurations of networks of processes. Conventions that allow newly created processes to be connected with already existing networks of processes need to be established. This can be achieved by establishing new channels via the ones it already has, e.g. by allowing some mechanism for sending channels along channels.

After introducing the idea of timestamping, there are two threads for further

research in deterministic concurrency:

- apply the timestamping approach in existing `GUI` libraries, imperative or not so that they can be made deterministic;
- find a purely functional explanation that combines the timestamping approach with the idea of independent demand.

## 12.3 Deterministic Distribution

In this thesis, based on the belief that the properties of functional languages allow programs to be easily distributed, we have presented a purely declarative model for distribution of computation based on the non-strict functional paradigm. Distribution is based on deterministic concurrency; a purely functional form of concurrency which extends the demand driven model of execution of functional languages into one with multiple independent demands without the need for non-determinism to appear at the user level. Since referential transparency is retained in this distributed model, distribution can be expressed in purely functional terms. Moreover, this scheme provides a new framework for computational mobility where data, code and computation can migrate dynamically according to demand.

A simulation of distribution has been presented that emulates distributed execution on a single processor and demonstrates the validity of distribution based on deterministic concurrency. This has been built in the Brisk run-time system and consists of a collection of Brisk machines which exchange information via messages. Each machine consists of heap allocated objects together with an index table to keep track of references from other machines. The complete set of heaps can be thought of a forming a single global memory. The index table decouples the local heap from the global distributed heap, provides a global address space where objects can be uniquely named and records global knowledge of the inter-processor aspects of the system.

### 12.3.1 Further Work

The prototype implementation presented in this thesis has been implemented from the `BKL` level as more work is needed in the Brisk compiler. Furthermore, the parts of the distributed heap reside at the same directory and thus, all access the same environment file. It needs further to be extended using `TCP/IP` sockets to support several processors cooperating in the distributed evaluation.

# Appendix A

# Algorithm for Deterministic Merging

```
module Process (
Process, Label(..), Code(..), Action(..), Event(..),
process, deliver, accept, step, showProcess, Status(..))
where

import Time
import List (transpose)

data Process =
  Proc ID Time Status Label Code State [Channel] [Channel]

instance Show Process where
   showsPrec p t s = showProcess t ++ s

getInChannels :: Process -> [Channel]
getInChannels (Proc _ _ _ _ _ _ ins _) = ins

getOutChannels :: Process -> [Channel]
getOutChannels (Proc _ _ _ _ _ _ _ outs) = outs
```

```
showProcess :: Process -> String
showProcess (Proc id t status label code st ins outs) =
   unwords ([id, show t, show status] ++ map show ins)

type ID = String
type Label = String
type Code = Data -> Label -> (Action, Label)
type Data = String
data Action = DoNothing | Receive | Send Int Data
type State = [(ID,Time)]

noData = "" :: Data

data Channel = Chan ID [Event]
instance Show Channel
   where showsPrec p t s = showChannel t ++ s

showChannel :: Channel -> String
showChannel (Chan _ es) = show es

enqueue :: Event -> Channel -> Channel
enqueue e (Chan id es) = Chan id (es ++ [e])

dequeue :: Channel -> (Event, Channel)
dequeue (Chan id es) = (head es, Chan id (tail es))

front :: Channel -> Event
front (Chan id es) = head es

empty :: Channel -> Bool
empty (Chan is es) = null es
```

```
data Status = Active | Waiting | Passive deriving Eq
instance Show Status
   where showsPrec p t s =  showStatus t ++ s

showStatus :: Status -> String
showStatus Active = "A"
showStatus Waiting = "W"
showStatus Passive = "P"

data Event = Message ID Time Data | Sync State

instance Show Event where
   showsPrec p t s = showEvent t ++ s

showEvent :: Event -> String
showEvent (Message id t d) = "M" ++ show t
showEvent (Sync ps) = process ps where
   process [p] = showItem p
   process (p:ps) = showItem p ++ "/" ++ process ps
   showItem (id,t) = id ++ show t

process :: ID->Status->Label->Code->[ID]->[ID]->Process
process id st s f is os =
  Proc id (time [1]) st s f [] ins outs where
   ins = [Chan id [Sync [("?", time [1])]] | id <- is]
   outs = [Chan id [] | id <- os]

deliver :: Int -> Event -> Process -> Process
deliver i e (Proc id t status label f state ins outs) =
   Proc id t status label f state ins1 outs where
   ins1 = insert (enqueue e (ins !! i)) i ins

insert y 0 (x:xs) = y:xs
```

```
insert y n (x:xs) = x : insert y (n-1) xs

accept :: Int -> Process -> (Event, Process)
accept i (Proc id t status label f state ins outs) =
   (event, Proc id t status label f state ins outs1) where
   outs1 = insert chan1 i outs
   chan = outs !! i
   (event, chan1) = dequeue chan

step :: Process -> Process
step proc = broadcast (execute (synchronise proc))

synchronise :: Process -> Process
synchronise (Proc id t a label f state is os) =
   Proc id t a label f state (map sync is) os where
   sync (Chan id es) = Chan id (sync1 es)
   sync1 (Sync _ : e : es) = sync1 (e:es)
   sync1 inp = inp

execute :: Process -> Process
execute proc@(Proc id t Active label f state ins outs) =
   perform act (Proc id t Active label1 f state ins outs)
   where
   (act, label1) = f noData label
execute proc@(Proc id t status label f state ins outs) =
  if not (isMessage msg)
  then Proc id t status label f state ins outs
  else
  perform act (Proc id t1 status label1 f state ins1 outs)
  where
  (msg, ins1) = receive ins
  Message _ _ dat = msg
  (act, label1) = f dat label
```

```
  t1 = clock t
  isMessage :: Event -> Bool
  isMessage (Message _ _ _) = True
  isMessage _ = False

perform :: Action -> Process -> Process
perform act (Proc id t status label f state ins outs) =
  case act of
   DoNothing -> Proc id t Active label f state ins outs
   Receive -> Proc id t Waiting label f state ins outs
   Send i x ->
    Proc id t1 Active label f state ins outs1
  where
  outs1 = putMsg i t x 0 outs
  putMsg i t x j [] = []
  putMsg i t x j (out:outs) =
   (if i==j then enqueue (Message id t x) out else out) :
   putMsg i t x (j+1) outs
  t1 = tick t

broadcast :: Process -> Process
broadcast (Proc id t status label f state0 ins outs) =
   Proc id t status label f state0 ins outs1
   where
   state1 =
    mergeAll [state | Chan _ list <- ins, Sync state <- list]
   state2 = adjust id status t state1
   outs1 = if state2 == state0 then outs else map put outs
   put out =
     if empty out then enqueue (Sync state2) out else out

merge :: State -> State -> State
merge ps1 [] = ps1
```

```
merge [] ps2 = ps2
merge ((id1,t1):ps1) ((id2,t2):ps2) =
  if id1 < id2 then (id1,t1) : merge ps1 ((id2,t2):ps2) else
  if id1 > id2 then (id2,t2) : merge ((id1,t1):ps1) ps2 else
  (id1, min t1 t2) : merge ps1 ps2

mergeAll :: [State] -> State
mergeAll xs = foldr merge [] xs

adjust :: ID -> Status -> Time -> State -> State
adjust id Active t _ = [(id,t)]
adjust id st t [] = []
adjust id st t ((id1,t1):ps1) | id1 == id =
   adjust id st t ps1
adjust id st t ((id1,t1):ps1) =
   (id1, t2) : adjust id st t ps1 where
   t2 = if st /= Waiting || t1 >= t then clock t1 else t

receive :: [Channel] -> (Event,[Channel])
receive ins = (head evts, ins1) where
 (mint,mini) =
   minimum [(channelTime ch, i) | (ch,i) <- zip ins [0..]]
 Chan id evts = ins !! mini
 inps1 = case head evts of
  Message id t dat ->
   insert (Chan id (Sync [(id, tick t)]:tail evts)) mini ins
  Sync t -> ins

channelTime :: Channel -> Time
channelTime (Chan _ queue) = check queue where
   check (Message id t _ : _) = t
   check (Sync ps : _) = minimum (map snd ps)
```

# Bibliography


[1] Peter Achten and Rinus Plasmeijer. The ins and the outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[2] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge Mass, 1986.

[3] Gert Akerholt, Kevin Hammond, Simon L. Peyton Jones, and Phil Trinder. Processing Transactions on GRIP. In *Parallel Architectures and Languages Europe*, June 1993.

[4] Gregory Andrews. *Concurrent Programming.* Addisson Wesley, 1991.

[5] L. Augustsson, B. Boutel, F.W. Burton, J. Fairbairn, J. H. Fasel, A. D. Gordon, M. Guzman, R.J.M. Hughes, P. Hudak, T. Johnson, M.P. Jones, R. Kieburtz, R. Nikhil, W.D. Partain, E. Meijer, S. L. Peyton Jones, and P.L. Wadler. *Report on the Functional Programminf Language Haskell, Version 1.4*, 1997.

[6] Lennart Augustsson. *Compiling lazy functional languages, part II.* PhD thesis, Chalmers University, 1987.

[7] Henri Bal, Jennifer Steiner, and Andrew Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[8] H. P. Barendregt. *The Lamdba Calculus, its syntax and semantics.* North Holland, Amsterdam, The Netherlands, 1984.

[9] M. Ben-Ari. *Principles of Concurrent Programming.* Prentince Hall, 1982.

[10] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentince Hall, 1990.

[11] Kenneth Birman, André Schipper, and Pat Stephenson. Lightweight Causal and Atomic Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[12] Sylvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. Eden-Language Definition and Operational Semantics. Technical Report Technical Report 96-10, Philipps Universität Marburg, 1996.

[13] Sylvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. DREAM:The DistRibuted Eden Abstract Machine. In Chris Clack, Tony Davie, and Kevin Hammond, editors, *9th International Workshop on the Implementation of Functional Languages*, volume 1467 of *LNCS*, pages 250–269, Saint-Andrews, Scotland, September 1997. Springer Verlag.

[14] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Pitman, 1991.

[15] Geoffrey Burn and Simon L. Peyton Jones. The Spineless G-machine. In *Conference on Lisp and Functional Programming*, 1988.

[16] W. F. Burton. Nondeterminism with Referential Transparency in Functional Programming Languages. *The Computer Journal*, 31(3):243–247, 1988.

[17] Mary Campione and Kathy Walrath. *The Java Tutorial: Object-Oriented Programming for the Internet.* Amazon, 1998.

[18] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[19] Luca Cardelli. Mobile Computation. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems*, volume 1222 of *LNCS*, Linz, Austria, July 1996. Springer Verlag.

[20] Luca Cardelli. Global Computation. *Sigplan Notices*, Position Paper, 1997.

[21] M. Carlsson and T. Hallgren. Fudgets - A Graphical User Interface in a Lazy Functional Language. In *Functional Programming and Computer Architecture*, Copenhagen, Denmark, March 1993. ACM Press.

[22] N. Carriero and D. Gelernter. Linda in Context. *CACM*, 32(4):444–458, 1989.

[23] David Carter. Deterministic Concurrency. Master's thesis, Department of Computer Science, University of Bristol, September 1994.

[24] Manuel M.T. Chakravarty. A Self-Scheduling, Non-Blocking, Parallel Abstract Machine for Lazy Functional Languages. Proceedings of the 6th International Workshop on the Implementation of Functional Languages, 1994.

[25] Manuel M.T. Chakravarty and Hendrik C.R. Lock. The JUMP-machine: a Generic Basis for the Integration of Declarative Paradigms. Proceedings of the Post-ICLP '94 Workshop, 1994.

[26] Koen Claessen. ST in pure Haskell. Haskell Mailing List, July 1998.

[27] W. Clinger. Nondeterministic call by need is neither lazy nor by name. In *Conference on Lisp and Functional Programming*, 1982.

[28] E.C. Cooper and Morrisett J.G. Adding Threads to Standard ML. Technical Report Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1984.

[29] David E. Culler, Seth Copen Goldstein, Schauser, Klaus Erich Schausen, and Thorsten von Eicken. TAM-a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, 1993.

[30] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[31] Chris Dornan. *Type-Secure Meta-Programming*. PhD thesis, Bristol University, 1998.

[32] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, Schauser, and Klaus Erich Schausen and. Active messages: a mechanism for integrated communication and computation. In *19th International Symposium of Computer Architecture*. ACM Press, May 1992.

[33] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.

[34] S. Finne and Simon L. Peyton Jones. Programming Reactive Systems in Haskell. In *Glasgow Functional Programming Workshop*, Ayr, Scotland, 1994. Springer-Verlag.

[35] S. Finne and Simon L. Peyton Jones. Composing Haggis. In *Proc 5th Eurographics Workshop on Programming Paradigms in Graphics*, Maastricht, September 1995.

[36] S. Finne and Simon L. Peyton Jones. Concurrent Haskell. In *Principles of Programming Languages*, pages 295–308, St Petersburg Beach, Florida, January 1996. ACM Press.

[37] Mattern Friedmann. Virtual Time and Global States of Distributed Systems. In Cosnard M. et al., editor, *Proceedings of the International Workshop of Parallel and Distributed Algorithms*, pages 215 – 226. Elsevier, 1989.

[38] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[39] N.H. Gehani and T.A. Cargill. Concurrent Programming in Ada. *Software Practice and Experience*, 14(5):413–427, May 1984.

[40] A Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. In *TAPSOFT '89 (vol 2)*, volume 352 of *LNCS*, pages 184–209, New York, N.Y., March 1989. Springer Verlag.

[41] J. Gosling, J. B. Joy, and G. Steele. *The Java Language Specification*. Addisson Wesley, 1996.

[42] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type Classes in Haskell. In *ESOP*, January 1994.

[43] K Hammond and Simon L. Peyton Jones. Profiling Scheduling Strategies on the GRIP Parallel Reducer. In *Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages*, 1992.

[44] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming Languages for Distributed Applications. *New Generation Computing*, 16(3):223–261, 1998.

[45] Seif Haridi, Peter Van Roy, and Gert Smolka. An Overview of the Design of Distribyted Oz. In *PASCO'97, Second International Symposium on Parallel Symbolic Computation*, New York, US, 1997.

[46] D. Harrison. RUTH: a functional language for real-time programming. In J.W. Bakker, A.J. Nijman, and P.C. Treleaven, editors, *PARLE II*, volume 259 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 1987. Springer Verlag.

[47] P. Henderson. Purely Functional Operating Systems. In J. Darlington, P. Henderson, and D. A. Tunrer, editors, *Functional Programming and its Applications*. Cambridge University Press, 1982.

[48] C. A. R. Hoare. *Communicating Sequential Processes*. Prentince Hall, 1985.

[49] C.A.R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, August 1978.

[50] Ian Holyer. *Functional Programming with Miranda*. UCL Press, 1991.

[51] Ian Holyer and David Carter. Concurrency in a Purely Declarative Style. In John T. O' Donnell and Kevin Hammond, editors, *Glasgow Functional Programming Workshop*, pages 145–155, Ayr, Scotland, 1993. Springer-Verlag.

[52] Ian Holyer, Neil Davies, and Chris Dornan. The Brisk Project: Concurrent and Distributed Functional Systems. In *Glasgow Functional Programming Workshop*, Ullapool, July 1995. Springer-Verlag.

[53] Ian Holyer, Neil Davies, and Eleni Spiliopoulou. Distribution in a Demand Driven Style. In *1st International Workshop on Component-based software development in Computational Logic*, September 1998.

[54] Ian Holyer and Eleni Spiliopoulou. The Brisk Machine: A Simplified STG Machine. In Chris Clack, Tony Davie, and Kevin Hammond, editors, *9th International Workshop on the Implementation of Functional Languages*, volume 1467 of *LNCS*, pages 20–38, Saint-Andrews, Scotland, September 1997. Springer Verlag.

[55] Ian Holyer and Eleni Spiliopoulou. Concurrent Monadic Interfacing. In K. Hammond, A.J.T. Davie, and C. Clack, editors, *10th International Workshop on the Implementation of Functional Languages (IFL '98)*, volume 1595 of *LNCS*, pages 73–89, London, United Kingdom, September 1998. Springer Verlag.

[56] Ian Holyer and Eleni Spiliopoulou. Deterministic Communications. Technical report, University of Bristol, Department of Computer Science, January 1999.

[57] Lucent Technologies Inc. *Inferno Programmers Guide*. Murray Hill, NJ, 1997.

[58] Thomas Johnsson. Lambda lifting: Transforming Programs to Recursive Equations. In Jouannaud, editor, *Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–205. Springer Verlag, 1985.

[59] Thomas Johnsson. *Compiling lazy functional languages.* PhD thesis, Chalmers University, 1987.

[60] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.

[61] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, December 1993.

[62] S. B. Jones and F. Sinclair. Functional Programming and Operating Systems. *The Computer Journal*, 32(2):162–174, 1989.

[63] Koji Kagawa. Compositional references for stateful functional programs. In *ACM SIGPLAN, International Conference of Functional Programming*, Amsterdam, The Netherlands, June 1997. ACM Press.

[64] H. Kingdon, D.R. Lester, and G.L. Burn. The HDG-machine: a highly distributed graph-reducer for a transputer network. *The Computer Journal*, 34(4):290–301, Aug 1991.

[65] Steve Kleiman, Devanag Shah, and Bart Smaalders. *Programming with Threads*. Prentince Hall, 1996.

[66] F.C. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1995.

[67] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–570, July 1978.

[68] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 1(5):1–11, February 1987.

[69] J. Launchbury and Simon L. Peyton Jones. Lazy Functional State Threads. In *Programming Languages Design and Implementation*, Orlando, 1994. ACM Press.

[70] John Launchbury. Lazy Imperative Programming. In *ACM SigPlan Workshop on State in Programming Languages*, June 1993.

[71] Paulson L.C. *ML for the Working Programmer*. Cambridge University Press, New York, N.Y., 1991.

[72] David Lester and Simon L. Peyton Jones. A modular fully-lazy lambda lifter in haskell. *Software Practice and Experience*, 21(5), 1991.

[73] D.R. Lester. An efficient distributed garbage collection algorithm. In E Odijk, M. Rem, and Syte J.C., editors, *Parallel Architectures and Languages Europe*, volume 365 of *LNCS*, pages 207–223, Eindhoven, The Netherlands, June 1989. Springer Verlag.

[74] D.R. Lester. An efficient distributed garbage collection algorithm. In *Functional Programming and Computer Architecture*, volume Lecture Notes in Computer Science, pages 207–223, London, UK, September 1989. Springer Verlag.

[75] Inmos Limited, editor. *Occam programming manual*. Prentice-Hall, 1984.

[76] J. W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[77] Umesh Maheshwari and Barbara Liskov. Collecting Cyclic Distributed Garbage by Controlled Migration. *Distributed Computing*, 10(2):79–86, 1997.

[78] Friedmann Mattern. Distributed control algorithms. In F. Oezguner and F. Ercal, editors, *Parallel Computing on Distributed Memory Multiprocessors*, volume Functional Programming and Computer Architecture, pages 167–185. ACM Press, 1993.

[79] Friedmann Mattern and S. Fuenfrocken. A non-blocking lightweight implementation of causal order message delivery. In A.Schiper K.P.Birman, F.Mattern, editor, *Theory and Practice in Distributed Systems*, volume 938 of *LNCS*, pages 197–213. Springer Verlag, 1995.

[80] David May and Henk L Muller. Icarus language definition. Technical Report CSTR-97-007, Department of Computer Science, University of Bristol, January 1997.

[81] J. McCarthy. A basic mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70, North-Holland, Amsterdam, 1963.

[82] Robin Milner. *Communication and Concurrency*. Prentince Hall, 1989.

[83] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I,II. *Information and Computation*, 100(1):1–77, 1992.

[84] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Logic in Computer Science Conference*, 1989.

[85] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 1990.

[86] Flemming Nielson. *ML With Concurrency*. Springer Verlag, 1996.

[87] Paddy Nixon, Vinny Cahill, and Fethi Rabhi. Building Parallel and Distributed Systems. *The Computer Journal*, 40(8):463–464, 1997.

[88] Rob Noble and Colin Runciman. Gadgets: Lazy Functional Components for Graphical User Interfaces. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *PLILP'95: Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, volume 982 of *LNCS*, pages 321–340. Springer-Verlag, Sept 95.

[89] Adrian Nye. *Xlib Programming Manual*. O' Reilly and Associates, Inc, 1990.

[90] John O' Donnell. Dialogues: a basis for constructing programming environments. In *ACM Symposium on Language Issues in Programming Environments*, pages 19–27, Seattle, 1985. ACM Press.

[91] Department of Defence. *Reference Manual for the Ada Programming Language*. Murray Hill, NJ, 1997.

[92] Hűseyin Pehlivan and Ian Holyer. A Recovery Mechanism for Shells. Technical report, University of Bristol, 1998.

[93] Alastair Penney. *Augmenting Trace-based Functional Debugging*. PhD thesis, Bristol University, September 1999.

[94] L. Peyton Jones, Simon and Simon Marlow. The New GHC/Hugs Runtime System. In *Implementation of Functional Languages*, Draft Proceeedings, London, UK, September 1998.

[95] L. Peyton Jones, Simon and Simon Marlow. The New GHC/Hugs Runtime System. http://research.microsoft.com/Users/simonpj/, 1998.

[96] S. L. Peyton Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, 32(2):175–187, 1989.

[97] S. L. Peyton Jones, J. Launchbury, M.B. Shields, and A.P. Tolmach. Bridging the gulf: a common intermediate language for ml and haskell. In *Principles of Programming Languages*. ACM Press, 1998.

[98] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[99] Simon L. Peyton Jones. *Implementing Functional Languages*. Prentince Hall, 1992.

[100] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.

[101] Simon L. Peyton Jones, Chris Clack, and Jon Salkild. High-performance parallel graph reduction. In E Odijk, M. Rem, and Syte J.C., editors, *Parallel Architectures and Languages Europe*, volume 365 of *LNCS*, pages 193–206, Eindhoven, The Netherlands, June 1989. Springer Verlag.

[102] Simon L. Peyton Jones and J. Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[103] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming and Computer Architecture*, Sept 1991.

[104] Simon L. Peyton Jones and Erik Meijer. `Henk:` a typed intermediate language. In *Workshop on Types in Compilation*, Amsterdam, June 1997.

[105] Simon L. Peyton Jones and John Salkild. The Spineless Tagless G-machine. In MacQueen, editor, *Functional Programming and Computer Architecture*. Addisson Wesley, 1989.

[106] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *Principles of Programming Languages*, Charleston, January 1993. ACM Press.

[107] Benjamin Pierce and David Turner. Concurrent Objects in a Process Calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and*

*Practice of Parallel Programming*, volume 907 of *LNCS*, pages 235–252, Sendai, Japan, November 1994. Springer Verlag.

[108] Benjamin C. Pierce, Didier Rémy, and N. Turner, David. A typed higher-order programming language based on pi-calculus. In *In Workshop on Type theory and its Apllication to Computer Systems*, Kyoto University, July 1993.

[109] Landin. P.J. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1964.

[110] J.H. Reppy. *Higher-order Concurrency.* PhD thesis, Cornell University, 1991.

[111] John Reppy. First-class Synchronous Operations. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming*, volume 907 of *LNCS*, pages 235–252, Sendai, Japan, November 1994. Springer Verlag.

[112] Pascal Serrarens. BriX - A Deterministic Concurrent Functional X Windows System. Master's thesis, Computer Science Department,Bristol University, June 1995.

[113] Jon Siegel. *CORBA Fundamentals and Programming.* John Wiley and Sons, 1996.

[114] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts.* Addisson Wesley, 1998.

[115] Gert Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science, vol. 1000*, pages 324–343. Springer Verlag, Berlin, 1995.

[116] H. Søndergaard and P. Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27(6):505–517, 1990.

[117] H. Søndergaard and P. Sestoft. Non-determinism in Functional Languages. *The Computer Journal*, 35(5):514–523, 1992.

[118] Eleni Spiliopoulou. An Analogy between Client-Server, RPC's and a Functional Deterministic Distributed System. In *Spring Research Conference*, Bristol, April 1997. Faculty of Engineering.

[119] Eleni Spiliopoulou, Ian Holyer, and Neil Davies. Distributed Programming, a purely functional approach. In *International Concference of Functional Programming*, Amsterdam, The Netherlands, June 1997. ACM Press. Poster Abstract.

[120] Joseph E. Stoy. *Denotational Semantics: the Scott-Stratchey Approach to Programming Language Theory.* MIT Press, 1977.

[121] W.R. Stoye. A new scheme for writing functional operating systems. Technical Report Technical Report 56, Cambridge University, Computer Lab, 1984.

[122] Andrew S. Tanenbaum. *Structured Computer Organisation.* Prentice Hall, 1999.

[123] Paul Tarau and Veronica Dahl. Mobile Threads through First Order Continuations. In *Proceedings of APPAI-GULP-PRODE'98*, Coruna, Spain, July 1998.

[124] B. Thomsen, L. Leth, S. Prased, T.-M. Kuo, A. Kramer, F.C. Knabe, and A. Giacalone. Facile Antigua Release Programming Guide. Technical Report ECRC-93-20, European Computer Industry Research Centre, Munich, Germany, Dec, 1993.

[125] Tommy Thorne. Programminf Languages for Mobile Code. *ACM Computing Surveys*, 29(3):212–239, September 1997.

[126] Philip Trinder, Kevin Hammond, J. Mattson, A. Partridge, and Simon L. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Programming Languages Design and Implementation*, Philadelphia, USA, 1996. ACM Press.

[127] D.A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[128] David Turner. Functional Programming and Communicating Processes. In J.W. Bakker, A.J. Nijman, and P.C. Treleaven, editors, *PARLE II*, volume 259 of *LNCS*, pages 54–74, Eindhoven, The Netherlands, June 1987. Springer Verlag.

[129] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile Objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.

[130] Philip Wadler. Comprehending Monads. In *Conference on Lisp and Functional Programming*, pages 61–78, June 1990.

[131] Philip Wadler. Monads for functional programming. *Lecture notes for Marktoberdorf Summer School on Program Design Calculi, Springer-Verlag*, Aug 1992.

[132] Philip Wadler. The essence of functional programming. In *Principles of Programming Languages*, Albuquerque (New Mexico, USA), January 1992. ACM Press.

[133] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Principles of Programming Languages*. ACM Press, Jan 1989.

[134] C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus*. PhD thesis, Oxford University, 1971.

[135] Claes Wikstrom. Distributed Programming in Erlang. In *PASCO'94, First International Symposium on Parallel Symbolic Computation*, Linz, Austria, September 1994.

[136] Stuart Wray and J. Fairbairn. Tim - a simple lazy abstract machine to execute supercombinators. In *Functional Programming and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, Portland, Oregon, 1987. Springer Verlag.

[137] Shao Zhong. An Overview of the FLINT/ML Compiler. In *Workshop on Types in Compilation*, Amsterdam, June 1997.

# Index